Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

А. В. Зафиевский
А. А. Короткин
А. Н. Лататуев

Базы данных

Учебное пособие

Рекомендовано
Научно-методическим советом университета
для студентов, обучающихся по направлениям
Прикладная информатика (в экономике)
и Прикладная информатика

Ярославль 2012

УДК 004.65(075.8)
ББК З973.233-018.2я73
З 37

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2012 года

Рецензенты:
Г. П. Штерн, кандидат технических наук, доцент;
кафедра прикладной математики
Ярославского государственного технического университета

З 37 **Зафиевский, А. В. Базы данных:** учебное пособие / А. В. Зафиевский, А. А. Короткин, А. Н. Латаtуев; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2012. – 164 с.

ISBN 978-5-8397-0860-0

В пособии дается общий обзор различных типов баз данных и соответствующих методов их использования. Описываются реляционная модель баз данных, основные конструкции языка SQL, методы проектирования структуры баз данных, основанные как на принципах нормализации, так и на использовании модели «сущность-связь». Приведены основные понятия о транзакциях в базах данных. В заключение рассмотрен общий подход к технологиям «клиент-сервер».

Предназначено для студентов, обучающихся по направлениям 080801.65 Прикладная информатика (в экономике), 230700.62 Прикладная информатика (дисциплина «Базы данных», цикл Б3, блок ОПД), очной формы обучения.

УДК 004.65(075.8)
ББК З973.233-018.2я73

ISBN 978-5-8397-0860-0

Оглавление

Предисловие

Умение грамотно работать с современными базами данных является одним из ключевых требований к любому специалисту в области компьютерных технологий. Важную роль в освоении технологий баз данных играет понимание их теоретических основ. Предлагаемое учебное пособие нацелено именно на ознакомление студентов с фундаментальными понятиями баз данных. В нем дается общий обзор различных типов баз данных и соответствующих методов их использования. Особое внимание уделяется наиболее распространенным реляционным базам данных. Описываются основные конструкции языка SQL, являющегося фактическим стандартом работы с реляционными базами данных. Приведены методы проектирования структуры баз данных, основанные как на принципах нормализации, так и на использовании модели «сущность-связь». Отдельная глава посвящена описанию транзакций, являющихся основным средством обеспечения надежной работы с базами данных. В заключение рассматривается общий подход к технологиям «клиент-сервер».

Разумеется, в небольшом пособии невозможно охватить все аспекты теории (не говоря уже о практике) баз данных. Пособие отражает скорее опыт авторов в преподавании этой дисциплины (и число выделяемых часов на лекции). Приведенный список литературы поможет значительно расширить кругозор знаний о базах данных. Стоит отметить в этом списке фундаментальный труд одного из основоположников реляционной модели К. Дейта [1], а также используемый во многих зарубежных университетах курс Г. Гарсиа-Молины и др. [2]. Из отечественных изданий стоит отметить учебники С. Кузнецова [4, 5] и А. Маркова, К. Лисовского [7]. Работа с наиболее распространенными серверами баз данных описывается в книге А. Кригеля и Б. Трухнова [9]. И конечно же, «море информации», как провозглашается на сайте CITForum.ru, содержится на бескрайних просторах Интернета.

Главы 1, 3, 6 этого пособия написаны А. В. Зафиевским, главы 2, 4, 5 – А. А. Короткиным, глава 7 – А. Н. Лататуевым. Авторы искренне надеются, что учебное пособие окажется полезным всем изучающим огромный и увлекательный мир баз данных, а об имеющихся ошибках им будет сообщено.

1. Что такое базы данных

1.1. Понятие базы данных

Базы данных играют огромную роль в современном информационном мире. Очень часто базами данных называют любые массивы информации, записанные на электронные носители или получаемые по компьютерным сетям. Это, однако, не совсем так. Передача информации существовала и в докомпьютерную эпоху, но именно компьютерные технологии позволили проводить обработку громадных объемов информации, которая была невозможна без применения компьютеров. Именно этот факт имеют в виду, говоря о внедрении информационных технологий в современное производство и общественную жизнь. Можно сказать, что информационные технологии представляют собой методику, основанную на сборе максимальной информации о соответствующей области деятельности и ее компьютерной обработке. Информационной основой этих технологий и являются *базы данных*.

В самом общем виде можно сказать, что базы данных – это большие массивы информации о какой-либо сфере производственной или общественной деятельности, предназначенные для коллективного использования и допускающие компьютерную обработку этой информации.

С этой точки зрения обычные библиотеки, в частности, не являются базами данных, хотя и являются хранилищами большого количества информации, используемой различными потребителями. Даже при наличии электронного каталога библиотека не может предоставить электронные варианты всех хранящихся в ней материалов, что делает невозможной их автоматизированную компьютерную обработку.

Одним из непосредственных следствий приведенного определения является то, что база данных должна хранить служебную информацию о собственной структуре (*метаинформацию*). Именно на основе такой информации создаются компьютерные программы, обрабатывающие основную информацию базы данных.

Отметим, что данное определение базы данных, конечно же, не является полным. Сущность этого понятия будет раскрываться постепенно, по мере освоения различных аспектов работы с базами данных.

1.2. Окружение базы данных

Использование информационных технологий предполагает наличие соответствующей информационной системы, обеспечивающей их реализацию. Центральным элементом этой системы как раз и является база данных. Но, хотя она и является сердцем информационной системы, ее использование невозможно без других компонентов информационной системы. Эти компоненты являются достаточно независимыми друг от друга в том смысле, что структуру и состав каждого из них можно изменять, почти не затрагивая других компонентов. Основными компонентами окружения базы данных, составляющими информационную систему, являются:

- технические средства;
- математическое и программное обеспечение;
- организационное обеспечение;
- системные пользователи:
  - разработчики;
  - программисты;
  - администраторы;
  - операторы;
- конечные пользователи.

К *техническим средствам* (на компьютерном жаргоне – «железо») относятся компьютеры (пользовательские и серверы), средства телекоммуникации, а также периферийное оборудование (принтеры, сканеры видеокамеры и т. д.).

Математическое и программное обеспечение («софт») имеет гораздо более сложную структуру. В первую очередь к нему относятся, разумеется, алгоритмы, содержательно описывающие процессы обработки содержащейся в базе данных информации, а также прикладные программы, реализующие эти алгоритмы. Однако крайне важную роль играют программные средства, обеспечивающие доступ прикладных программ или даже непосредственно пользователей (реже – конечных, чаще – обслуживающего персонала) к базе данных. Совокупность этих обеспечивающих программ называется *Системой управления базой данных (СУБД)*. Их роль настолько важна, что иногда даже смешивают понятия базы данных и СУБД, говоря, например, «база данных Oracle», хотя более корректной была бы фраза «база данных, находящаяся под управлением СУБД Oracle». Важную роль в

комплексе программного обеспечения играет также используемая операционная система. Иногда совокупность используемых типов компьютеров, операционной системы и СУБД называют *платформой* базы данных.

Организационное обеспечение включает в себя различную документацию на информационную систему, а также регламенты (типовые процедуры), описывающие процессы создания (модификации) и эксплуатации системы. К сожалению, опыт создания многих информационных систем говорит о том, что именно эта составляющая оказывается наименее проработанной. Общеизвестно, что хорошие программисты не любят (и, по-видимому, не умеют) писать хорошую документацию (даже для себя). Создание документации – это сфера деятельности т. н. технических писателей, однако хорошие технические писатели встречаются намного реже, чем хорошие программисты. Тем не менее качество информационной системы во многом определяется качеством сопутствующей документации.

Особенно важную роль в документации играют регламенты, описывающие действия пользователей в штатных и нештатных (включая пожар, наводнение и землетрясение) ситуациях. Они должны быть такими, чтобы, с одной стороны, по возможности исключить творчество со стороны *обслуживающего персонала*, а с другой – предоставить максимальную простоту в работе *конечным пользователям*. Для достижения этих целей важно уже при разработке информационной системы предусмотреть необходимые действия по обучению всех категорий пользователей.

1.3. Базы данных первого поколения

Появление первых баз данных можно отнести к середине 50-х годов прошлого века. Их возникновение связано, в первую очередь, с деятельностью компании IBM, производившей вычислительные машины для коммерческих применений. Особенностью этих применений является то, что в результате их работы по сравнительно простым алгоритмам (суммирование, вычисление минимальных, средних и максимальных значений и т. п.) обрабатываются большие массивы данных, причем результаты обработки также имеют значительный объем и используются в дальнейшем другими приложениями. Все это привело к необхо-

димости, во-первых, уделять особое внимание методам хранения информации и эффективного доступа к ней (учитывая, что основным носителем больших объемов информации были магнитные ленты), а во-вторых, стандартизовать структуру данных с тем, чтобы вновь разрабатываемые приложения могли использовать эту структуру, не разрабатывая ее заново. Как мы уже отмечали, именно эти особенности характеризуют базу данных.

Низкая скорость обмена информацией между центральным процессором компьютера и внешней памятью (которая могла достигать нескольких минут), а также ограниченный объем основной памяти (несколько килобайт) привели к тому, что в базах данных хранилась только наиболее существенная информация, на основе которой могли быть подготовлены требуемые документы. Этим, кстати, объясняется само название «база данных» (DataBase), в котором слово «база» означает основу, фундамент, а не склад или хранилище, как принято обычно считать.

Заметим, что уже в этот период базы данных получили свое независимое материальное воплощение в виде бобины магнитной или перфоленты или колоды перфокарт, которые могли быть отчуждены от «родного» компьютера и перенесены на другой.

Важным этапом в развитии баз данных явилось создание в начале 1960-х годов той же фирмой IBM магнитных дисков, которые до сих пор являются основным носителем, на котором размещаются базы данных. Наиболее существенной их особенностью применительно к базам данных явилась возможность быстрого (миллисекунды) доступа к любым данным, хранящимся в произвольном месте на диске. Это дало возможность с помощью индексных файлов (или просто индексов) реализовать быстрый поиск информации в базе данных.

Вызванное этим усложнение программ, обеспечивающих работу с базами данных, и необходимость их унификации породило новый класс программного обеспечения – системы управления базами данных (СУБД). Первой коммерческой системой такого типа явилась система IMS (Information Management System) фирмы IBM. Данные в этой системе записываются в виде иерархической структуры, по существу повторяющей структуру взаимосвязей предметной области, для которой создается информационная система.

Дальнейшим развитием иерархических баз данных явились *сетевые* базы данных, устранившие один из недостатков иерар-

хических систем – сложность работы в случае, когда нижестоящие узлы имеют несколько вышестоящих. Финальным этапом в развитии баз данных этого типа можно считать отчет группы CODASYL DBTG, опубликованный в 1971 году. Пользуясь популярной в технологической сфере терминологией, можно отнести иерархические и сетевые базы данных к первому поколению. Основным способом работы с базами данных этого типа являлся т. н. «навигационный», заключающийся в перемещении по графу, описывающему структуру базы данных, и низкоуровневой поэлементной обработке содержащихся в узлах этого графа структур данных. Руководитель CODASYL DBTG, лауреат премии Тьюринга 1973 года Чарльз Бахман так и назвал свою тьюринговскую лекцию – «Программист как навигатор».

Основным недостатком баз данных первого поколения являлась сложность структуры данных даже для достаточно простых систем автоматизации управления, что приводило к сложности прикладных программ обработки данных, требующих детального описания навигационных путей к данным. Кроме того, наличие в программах навигационных путей крайне затрудняло модификацию структуры данных, поскольку требовало модификации всех соответствующих программ обработки. Это, в свою очередь, практически не давало возможности разделить функции проектировщика данных и проектировщика прикладных программ, что являлось заметным препятствием при реализации больших проектов.

1.4. Реляционные базы данных

Поворотным пунктом в теории и практике баз данных явился переход к реляционным базам данных. Начало этого этапа обычно датируется публикацией статьи сотрудника IBM Эдгара Кодда «Реляционная модель данных для больших совместно используемых банков данных», опубликованной в 1970 году. В этой и последующих статьях Э. Кодда и других авторов (среди которых следует отметить автора популярнейшего учебника по базам данных К. Дейта) были развиты основные положения реляционной модели данных. Интересно отметить, что само понятие «модель данных», означающее переход на более абстрактный уровень оперирования данными, было сформулировано именно Коддом и в базах данных первого поколения не использовалось.

Особенностью реляционной модели является совершенно новый взгляд на построение базы данных и действия с данными. Основной структурной единицей реляционной модели является *отношение* (relation), которое с теоретической точки зрения является легко описываемым и хорошо изученным математическим объектом, а с практической – может, с некоторыми оговорками, трактоваться как таблица простейшей структуры, не имеющая повторяющихся строк. Это резко отличает ее от иерархической и сетевой моделей данных, в которых узлы графа, описывающего структуру базы данных, могли содержать сколь угодно сложные информационные объекты. Упрощение структуры данных, тем не менее, не привело к потере функциональных возможностей обработки данных. Кодду удалось показать, что отношение (таблица) является в достаточной степени насыщенным информационным объектом, чтобы обеспечить произвольную обработку данных, если они изначально записаны в табличной форме. Простота структурных единиц реляционной модели позволила привлечь формальные математические методы для описания обработки данных. С этой целью Э. Коддом были разработаны языки *реляционной алгебры* и *реляционного исчисления*, обладающие необходимой полнотой, требуемой для такой обработки. Названные языки высокоуровневые, операндами и результатами в которых являются отношения, а детализация алгоритмов их реализации возложена на систему управления базой данных. Можно сравнить переход к реляционным базам данных с переходом от ассемблера к высокоуровневым языкам программирования. И кстати, одной из целей Кодда было предоставить конечным пользователям средства для работы с базами данных без обращения к услугам профессиональных программистов.

1.5. Язык SQL

Появление реляционной модели и смена идеологии баз данных привели к большому числу попыток практической реализации реляционной модели, как экспериментальных, так и коммерческих. Среди экспериментальных можно назвать System R корпорации IBM, из коммерческих – СУБД Oracle (Oracle Corporation) и DB2 (IBM). Эта деятельность привела к определенной корректировке реляционной модели и появлению нового языка

работы с реляционными данными – языка SQL (Structured Query Language). Первая редакция этого языка была опубликована в 1986 году и уточнена в 1989 году. Она обладала заметной неполнотой и в 1992 году была заменена новой редакцией SQL-92 (или SQL2). Вторая редакция появилась в период бурного роста числа автоматизированных систем, использующих базы данных, и явилась важным ориентиром для разработчиков СУБД. Совместимость с этим стандартом и сейчас выступает важной характеристикой той или иной системы управления базами данных.

Безусловно, расширение возможностей компьютерной техники и рост потребностей конечных пользователей порождают новые средства реализации этих потребностей со стороны систем управления базами данных. Это приводит к тому, что используемые в реальных системах языки работы с данными являются расширениями подмножеств языка SQL (кстати, то же самое происходит и с языками программирования). Критический анализ этих сокращений и расширений приводит к созданию новых редакций стандарта языка SQL. Так, за SQL-92 последовали SQL:1999 (SQL3), SQL:2003 и SQL:2008. Однако стоит заметить, что хотя реальные СУБД в ходе своего развития и приближаются к текущему стандарту, но не соответствуют ему, и для эффективного их использования все равно следует изучать документацию по конкретной СУБД. Тем не менее знание стандарта позволяет оценить перспективы развития СУБД и избежать решений, которые не будут впоследствии соответствовать стандарту и которые придется серьезно дорабатывать.

1.6. Системы управления базами данных

Как мы уже отмечали, важнейшую роль в программном обеспечении базы данных играет *система управления базой данных* (СУБД). Именно она определяет используемую модель данных, возможности многопользовательской работы, сервисные возможности и т. д. Опишем кратко основные функции, выполняемые системой управления базой данных.

Важнейшей из них является, безусловно, определение данных, реализующее принятую модель данных (в настоящее время – чаще всего реляционную), используемые типы данных, взаимосвязи между структурами данных. Кроме того, СУБД

обеспечивает отображение этой модели как пользователям в удобных для них форматах, так и на физические носители информации (магнитные диски). Тем самым формируются три уровня представления информации: концептуальный (или логический) – уровень проектировщика, внешний – уровень пользователей и физический – уровень хранения данных. Подобное разделение позволяет реализовать т. н. *независимость данных*, когда каждый уровень можно создавать или изменять относительно независимо от других. Это позволяет, например, проектировать концептуальный уровень «с запасом», ограничиваясь небольшим числом внешних представлений в расчете на будущее расширение функций системы. С другой стороны, в начале развертывания системы можно ограничиться небольшим объемом дисковой памяти, в дальнейшем наращивая его. СУБД скрывает механизм размещения данных на физических носителях, позволяя проектировщикам и пользователям сосредоточиться на основной сущности решаемых задач. Вместе с тем, несмотря на используемые СУБД алгоритмы оптимизации, она, как правило, позволяет проводить и «ручную» настройку размещения и обработки данных.

Второй фундаментальной функцией СУБД является обеспечение действий с данными (*манипулирование* данными). Основными операциями, обеспечиваемыми СУБД, являются поиск (отбор), вставка, удаление и замена данных. Реализация этих действий осуществляется либо непосредственно с помощью консольных интерфейсов, используемых системными или конечными пользователями, либо с помощью прикладных программ, использующих предоставляемые СУБД программные интерфейсы. Важно отметить, что с помощью тех же интерфейсов обеспечивается доступ к метаданным (называемым также системным каталогом, системными таблицами и т. п.), описывающим структуру базы данных. При этом СУБД должна гарантировать, что при любых действиях с базой данных не будет нарушена ее *целостность* (непротиворечивость хранимых данных).

К числу выполняемых СУБД действий с данными относится также преобразование их форматов для обмена данными с другими информационными системами.

Поскольку база данных, вообще говоря, предназначена для коллективного использования, к числу основных функций СУБД относится, во-первых, обеспечение доступа к базе данных по

компьютерной сети, а во-вторых, организация работы многих пользователей таким образом, чтобы они, по возможности, не мешали друг другу. Кроме того, при этом должно быть обеспечено *разграничение пользователей*, означающее создание групп пользователей, которым предоставляется различная информация и различные возможности действий с этой информацией.

Основным механизмом, обеспечивающим многопользовательскую работу с базой данных, являются *транзакции*. Транзакция представляет собой совокупность действий с группой данных, содержащихся в базе данных, являющихся, с точки зрения того или иного пользователя, единым действием. Типичным примером является продажа билетным кассиром одного или нескольких билетов привередливому пассажиру. В ходе этой операции пассажир может много раз поменять свои запросы, и законченной эта транзакция может считаться только тогда, когда он окончательно отойдет от кассы. Эффективное управление транзакциями является одним из важнейших показателей качества СУБД.

Тот же самый механизм транзакций применяется и в том случае, когда «пользователем» является окружающая среда. СУБД должна обеспечивать восстановление функционирования информационной системы после неблагоприятных воздействий (отказа аппаратуры, фатальных программных ошибок, некорректных действий пользователей, атаки злоумышленников и т. д.) таким образом, чтобы это не имело необратимых последствий для других пользователей.

Последней важной характеристикой СУБД, которую стоит упомянуть, является наличие большого количества вспомогательных программ (утилит). Их набор может быть весьма широк и разнообразен. К числу обязательных следует причислить средства массовой начальной загрузки (миграции) данных. Это связано с тем, что, как правило, при внедрении новой системы большие массивы информации уже накоплены и задача помещения их в базу данных может оказаться весьма сложной. К обязательным утилитам можно также отнести средства наблюдения за работой (мониторинга) системы и ее настройки на оптимальную работу. Очень часто в число вспомогательных утилит включают средства разработки, обеспечивающие моделирование данных и бизнес-процессов, высокоуровневые языки программирования, ориентированные на базы данных, и т. д.

1.7. Объектно ориентированные базы данных

Реляционная модель данных резко упростила базовые «кирпичики», из которых строится структура базы данных, используя в этом качестве почти исключительно таблицы. При этом, однако, усложнилась проблема учета взаимосвязей между элементами этой структуры. И хотя эта проблема в реляционной модели решается за счет использования внешних ключей, при большом количестве связей структура базы данных становится сложной для восприятия и неэффективной при выполнении запросов. Наиболее заметны эти недостатки в системах автоматизации проектирования (САПР), оперирующих сравнительной небольшим количеством информационных объектов сложной структуры. Иерархическая информация технических чертежей может включать десять и более уровней. При использовании реляционной модели для соответствующей информационной реализации в этом случае могут потребоваться десятки взаимосвязанных таблиц, причем эти таблицы будут содержать всего по нескольку строк, а некоторые – даже только по одной.

Эта проблема была менее заметна в базах данных первого поколения, поскольку в них допускались достаточно сложные базовые элементы и их иерархическая организация, однако программирование в этих базах данных было низкоуровневым, что не позволяло использовать их при реализации сложных проектов.

Сформировавшаяся к середине 1980-х годов идеология объектно ориентированного программирования привела к созданию модели объектно ориентированных баз данных (ООБД) и соответствующих СУБД. В этой модели информационные образы реальных объектов предметной области записываются в базу данных наиболее естественным образом, с сохранением их целостности и связей между ними. Кроме того, в базу же данных записываются функции («методы»), описывающие взаимодействие между объектами. Это заметно отличает ООБД от реляционных баз данных. В прикладных информационных системах, основывающихся на реляционных БД, существует принципиальный разрыв между структурной и процедурной частями. Структурная часть системы поддерживается всем аппаратом СУБД, ее можно моделировать, верифицировать и т. д., а система обработки данных создается независимо, и средства ее создания не являются неотъемлемой

частью СУБД. В объектно ориентированных же базах данных процессы совместного моделирования и гарантирования согласованности структурной (статической) и процедурной (динамической) частей интегрированы между собой. Тем самым язык манипулирования данными является встроенным компонентом объектно ориентированной системы. Можно сказать, что объектно ориентированная система БД представляет собой объединение системы программирования и СУБД. В последнее время расширяется использование в качестве базового языка программирования Java.

В 1990-е годы была развернута работа по стандартизации объектно ориентированного подхода к организации баз данных. Консорциумом ODMG поставщиков ООСУБД был выпущен стандарт ODMG-93, включающий в себя средства для построения законченного приложения, которое будет работать (после перекомпиляции) в любой совместимой с этой спецификацией ООСУБД. Впоследствии группой ODMG был выпущен финальный стандарт ODMG 3.0 (2000), включающий полную объектную модель данных и привязку к ней языков Smalltalk, C++ и Java. Позднее привязка к Java была замещена средствами Java Data Objects (JDO). В 2001 году группа ODMG была распущена ввиду завершения поставленных перед ней задач.

Наибольшее применение ООБД находят в областях, характеризующихся сложной структурой используемых данных. Это уже упоминавшиеся системы автоматизации проектирования, геоинформационные системы, интернет-приложения, использующие сложные мультимедийные структуры, и т. д.

Вместе с тем следует отметить, что, наряду с развитием объектно ориентированных БД, реляционные базы данных продолжают оставаться основным типом БД, обладая рядом преимуществ, среди которых:

- наличие серьезной и сбалансированной теоретической основы, которая, вместе с тем, достаточно проста и не требует длительного освоения;
- высокая скорость при работе с большими объемами данных;
- эффективная система разграничения полномочий и обеспечения доступа пользователей к данным.

Кроме того, во всем мире значительные средства уже инвестированы в реляционные СУБД и владельцы этих систем этих

систем заинтересованы в приобретении некоторых объектно ориентированных возможностей для установленных систем без их коренной ломки.

Поэтому более предпочтительным является комбинированный подход, который позволил бы воспользоваться достоинствами объектных баз данных, не отказываясь полностью от уже используемых реляционных БД. Одним из вариантов в этом случае является использование *объектно реляционных* БД, представляющих собой реляционные БД с объектными возможностями. Еще более консервативный способ – использование объектных расширений для установленных БД. Эта идеология нашла отражение уже в стандарте SQL:1999 и используется ведущими поставщиками реляционных систем управления базами данных.

1.8. Полнотекстовые базы данных

В классических базах данных хранится лишь небольшая часть данных, образующая основу всей информационной структуры, вследствие чего жестко нормируются как формат данных, так и взаимосвязи между ними. Вместе с тем дальнейшее развитие компьютерной техники привело к тому, что стало возможным хранить на компьютерных носителях не только базовую информацию, но и полные текстовые документы в виде неструктурированных текстов или даже их фотографических изображений, а также информационные объекты других типов: рисунки, анимацию, аудио- и видеозаписи. В результате возникли большие хранилища данных, основным содержанием которых является изначально неструктурированная информация.

Вместе с тем хранящиеся объекты имеют сложную внутреннюю структуру, которая, однако, неизвестна в момент их помещения в хранилище. Поскольку основной задачей в подобных системах является отыскание объекта по каким-либо признакам, эти системы стали называться *информационно-поисковыми системами* (ИПС). Впоследствии на них также было распространено название «базы данных», несмотря на то что они содержат не только базовую, но и другую информацию. В связи с этим иногда различают классические базы данных, называя их ориентированными на данные, и базы данных с неструктурированной информацией, называя их ориентированными на документы или полно-

текстовыми. При этом под документом может пониматься не только текстовый документ, но и любой большой информационный объект (изображение, аудио- или видеоклип, компьютерная программа и т. д.), структура которого заранее не известна информационной системе. Разумеется, современные базы данных сочетают оба подхода, поэтому имеет смысл говорить лишь об их преимущественной направленности.

Проблематика полнотекстовых баз данных заметно отличается от таковой в структурированных базах данных. Прежде всего, хотя первоначально информация в исходных документах не является структурированной, запросы к базе данных на поиск нужного документа могут формироваться только на основе какой-либо информации о внутренней структуре хранящихся документов. Поэтому для работы с документами (далее для простоты будем рассматривать только текстовые документы) необходимы некоторые априорные представления о структуре имеющихся документов (модель документа). Простейшим примером может быть, например, представление документа как набора слов. В этом случае для поиска нужного документа можно указать перечень встречающихся в нем слов. Безусловно, любое подобное представление документов является упрощением, поэтому в информационно-поисковых системах как запросы, так и результаты выполнения запросов носят приблизительный характер.

Одной из основных задач в информационно-поисковых системах является автоматическая *индексация* документов, то есть выбор модели документов и организация указателей на документы (индексов), которые позволяют отбирать соответствующие запросу документы, не просматривая их целиком. Другой важной задачей является проблема *ранжирования* выдаваемых документов, то есть их выдача в порядке убывания степени соответствия запросу в соответствии с принятой моделью.

Наряду с описанным *дескрипторным* способом организации функционирования ИПС, широко используется также основной для бумажных библиотек *классификационный* способ, основанный на создании иерархической структуры, описывающей тематику документов (систематического указателя, или рубрикатора). Аналогом автоматической индексации в этом случае является автоматическая *рубрикация*, то есть размещение в узлах рубрикатора в результате

автоматического синтаксического анализа текста документа одного или нескольких указателей на этот документ.

В заключение стоит отметить, что информационно-поисковые системы представляют собой наиболее коммерциализированный продукт в сфере баз данных. В то время как другие разновидности баз данных обычно являются информационной основой какой-либо автоматизированной компьютерной системы, ИПС являются самодостаточными и непосредственно предоставляют услуги конечным пользователям. Это могут быть как простейшие системы, размещенные на оптическом диске, так и громадные научные и статистические базы данных, доступ к которым осуществляется через Интернет. Заметим, что одной из важнейших в этой сфере является проблема ценообразования, которая лежит за рамками настоящего пособия.

1.9. Слабоструктурированные базы данных

Развитие сети Интернет обострило проблему обмена информацией между различными информационными системами. Появилась возможность оперативной передачи данных из одной системы в другую, но в то же время в различных системах часто используются различные СУБД, различные форматы данных, различные системы кодирования и т. д. Для эффективного обмена данными потребовалось средство унифицированного обмена данными между различными информационными системами, которое позволило бы в основном сохранить имеющиеся информационные системы. Таким средством явился разработанный в конце 1990-х годов язык XML. Этот язык использует в качестве базовой иерархическую модель данных, дополненную некоторыми сетевыми возможностями. Основными блоками данных при работе с XML являются XML-документы, представляющие собой текстовые файлы, текст в которых размечен синтаксическими элементами языка XML, называемыми *тегами*. Часто данные, описываемые языком XML, называют слабоструктурированными, поскольку, с одной стороны, XML предъявляет достаточно строгие требования к описанию структуры данных, хранящихся в XML-документах, а с другой – допускает значительную свободу как в отношении типов данных, так и связей между элементами данных. Характерной особенностью XML является возможность отсутствия описания структуры данных XML-документа, по-

скольку эта структура в некотором стандартном виде может быть построена из самого текста XML-документа.

В настоящее время практически все СУБД имеют в своем составе средства, позволяющие выводить хранящуюся в них информацию в XML-формате. Хотя язык XML первоначально был создан для обмена данными через Интернет между различными информационными системами, впоследствии оказалось весьма удобным хранить всю информацию, относящуюся к системе, размещающейся на каком-либо веб-сайте, в виде совокупности XML-документов, формируя тем самым XML-базу данных. Более того, разработаны полноценные СУБД, хранящие информацию в формате XML (например, Oracle Berkeley DB или Tamino фирмы Software AG). Возможно, со временем такие системы будут использоваться более широко, хотя в настоящее время для хранения информации на веб-сайтах в основном используются классические реляционные базы данных.

1.10. Другие направления в организации и использовании баз данных

Остановимся кратко на направлениях развития баз данных, которые ранее не были затронуты.

Специфической разновидностью баз данных являются БД реального времени, применяемые в системах, в которых время реакции системы играет критическую роль. Это не означает, что выполнение запроса должно быть очень быстрым, но требуется, чтобы запрос был гарантированно выполнен за известное заранее время. Однако, поскольку подобные требования наиболее естественно выполнять в операционных системах реального времени (например, QNX), разработчики БД реального времени для массовых ОС пошли просто по пути резкого ускорения работы СУБД за счет размещения всей базы данных в оперативной памяти.

Так, компанией Oracle была разработана технология TimesTen и на ее основе – СУБД реального времени TimesTen In-Memory Database, легко интегрируемая с СУБД Oracle. В качестве другого примера можно привести СУБД RDM Mobile компании Birdstep.

Заметное развитие в последнее время получили базы данных с нетрадиционным наполнением: пространственными, временн***ы***ми и

мультимедийными данными. К этой же категории относятся и получающие все большее применение геоинформационные системы. Средства для работы с пространственными (геометрическими) данными включены, в частности, в последние версии СУБД от компаний Oracle и Microsoft. Что касается временны**х** баз данных, то это направление развивается гораздо медленнее, в основном из-за ограниченной потребности в специфических временны**х** задачах.

Мультимедийные базы данных получили широкое распространение в связи с расширением размеров сети Интернет и предоставляемых ей сервисов. Серьезной проблемой здесь является индексация хранящихся в базе данных аудио- и видео- «документов». В то время как для текстовых документов существует большое количество различных моделей автоматического формирования поискового образа документа, формирование поисковой модели для мультимедийных объектов представляется достаточно сложным. В этом направлении следует отметить деятельность по созданию стандарта MPEG-7. Обычно же организация мультимедийных БД выполняется по гораздо более простой схеме: для каждого «документа» вручную готовится текстовое описание и поиск ведется в базе данных этих описаний. Поскольку мультимедийные базы данных чаще всего являются коммерческими, то при их создании и эксплуатации возникает большое число проблем, связанных с доставкой содержимого пользователям и оценкой стоимости предоставленных услуг, однако эти проблемы чаще всего лежат за пределами основной тематики баз данных.

Накопление информации в базах данных, поддерживающих оперативную обработку производственной информации, привело к появлению нового класса БД – т. н. *хранилищ данных*, по существу представляющих собой электронные архивы предприятия. В отличие от систем оперативной обработки данных, хранящих информацию ровно столько времени, сколько нужно для ее обработки, хранилища данных накапливают всю информацию, обрабатываемую оперативными системами, проводят ее статистическую обработку и способны выдавать итоговую информацию с различных точек зрения («в различных разрезах»). Примыкающей к этому направлению областью исследований является область, обозначаемая термином, заимствованным из горнодобывающей отрасли, – «добыча данных» (data mining), более аккуратно называемая интеллектуальным анализом данных. Цель этой дея-

тельности – обнаружение скрытых в громадном объеме информации хранилищ данных сведений (знаний) на основе математических методов анализа данных: кластерного анализа, нейросетевых методов, генетических алгоритмов и т. д.

Еще одно направление в теории и практике баз данных – параллельные и распределенные системы. Общим для обоих типов систем является использование компьютеров, объединенных в сеть, причем сеть может быть как локальной, так и удаленной. Различной же является степень связи компьютеров между собой. В то время как в параллельных системах объединенные компьютерные ресурсы направляются на решение одной задачи, в распределенных системах, как правило, решаются разные задачи, использующие частично объединенные данные. Это приводит к тому, что организация взаимодействия между узлами сети в параллельных и распределенных системах оказывается принципиально различной. В частности, в распределенных системах чаще всего применяется технология *репликаций* (тиражирования), когда в различных узлах сети хранятся копии базы данных или ее частей и регулярно (и не всегда немедленно) проводится синхронизация содержимого этих копий. Технология репликаций позволила достаточно просто реализовать мобильные информационные системы, когда один или несколько узлов распределенной базы данных хранятся на переносных компьютерах (ноутбуках и т. п.). В определенные промежутки времени эти компьютеры могут подключаться к головной сети, в том числе с помощью беспроводного соединения, проводить синхронизацию информации и отключаться. В результате получается распределенная база данных, структура которой динамически изменяется во времени, оставаясь, тем не менее, полностью управляемой.

Повсеместное внедрение сети Интернет привело и к изменениям во взгляде на базы данных. Не говоря уже о том, что весь Интернет в целом подпадает под определение базы данных в широком смысле (совместно используемые данные с электронным описанием их структуры, поддерживаемым консорциумом W3C), стали появляться технологии, характерные для работы с базами данных, но применяемые при работе через Интернет. В первую очередь речь идет об интернет-сервисах, предоставляющих конкретные данные (прогноз погоды, курсы валют, программы

телепередач и т. п.), которые могут быть автоматически (без участия человека) получены и использованы.

Специфика Интернета возобновила интерес к базам данных, использующих в качестве входящей информации часто применяемые в этой сети *потоки данных*. По существу они представляют собой базы данных реального времени, но работающие не в локальной, а в глобальной сети.

В заключение упомянем такое явление, как *семантическая паутина* (Semantic Web), целью создания которой является реализация возможности автоматической обработки информации, доступной в Интернете. Средством для этого выступает формирование метаданных на специализированном языке OWL, однозначно характеризующих свойства и содержание ресурсов в Интернете по какой-либо предметной области, вместо используемого в настоящее время частотного и лексического анализа текстового содержимого документов. По существу это означает введение жестких структур в крайне слабо структурированную базу данных под названием «Интернет». Впрочем, Интернет развивается настолько быстро, особенно в части использования мультимедийного содержимого, что заявленная в 2001 году основателем Web Тимом Бернерсом-Ли концепция семантической паутины заметно устарела и либо будет серьезно переработана, либо вместо этого будут применяться другие средства, которые обеспечат не меньшие возможности.

Контрольные вопросы

1. Можно ли считать базой данных всемирную сеть Интернет? Обоснуйте ответ.
2. Из каких основных компонентов состоит информационная система, использующая базу данных?
3. Назовите основные модели данных.
4. Каковы основные функции системы управления базой данных?
5. Что такое транзакция?
6. Как можно оценить качество информационно-поисковой системы?

Практическое задание

Используя сеть Интернет, составьте перечень основных методов интеллектуального анализа данных.

2. Реляционная модель данных

Любая модель данных должна содержать три компонента:

1) ***структуру данных*** – описывает точку зрения пользователя на представление данных;
2) ***набор допустимых операций***, выполняемых на структуре данных. Модель данных предполагает, как минимум, наличие языка определения данных (DDL – Data Definition Language), описывающего структуру их хранения, и языка манипулирования данными (DML – Data Manipulation Language), включающего операции извлечения и модификации данных;
3) ***ограничения целостности*** – механизм поддержания соответствия данных предметной области на основе формально описанных правил.

В процессе исторического развития в СУБД использовались следующие модели данных: *иерархическая, сетевая, реляционная.*

В последнее время все большее значение приобретает объектно ориентированный подход к представлению данных.

В основе реляционной модели данных лежат три основные концепции: *реляционные отношения, ограничения целостности данных и алгебра реляционных операторов*.

2.1. Реляционное отношение

Рассмотрим следующий пример. Имеется список организаций-поставщиков, поставлявших некоторые товары в определенном количестве по определенной цене за 1 ед. Каждый поставщик описывается следующими характеристиками (свойствами, атрибутами): *код_п* (код поставщика), *имя_п* (имя поставщика), *гор* (город), в котором находится офис данного поставщика. Для описания товара используются такие характеристики, как *код_т* (код товара), *назв_ т* (название товара).

Всю информацию о поставках и свойствах объектов данной предметной области можно представить в виде таблицы, например, такого вида (см. табл. 2.1).

Таблица 2.1

Поставщик-товар

код_п	*имя_п*	*гор*	*код_т*	*назв_т*	*цена_1*	*кол_во*
п01	Скан	Ярославль	т14	монитор	7 500	10
п01	Скан	Ярославль	т09	принтер	3 400	4
п02	Альфа	Москва	т14	монитор	7 200	6
п03	Тензор	Ярославль	т03	HD диск	4 300	12
п11	ИП Иванов С. Н.	Тутаев	т23	стол	2 000	3

Имена столбцов таблицы – сокращенные названия характеристик, или атрибутов, объектов рассматриваемой предметной области (поставщиков и товаров), а также свойств поставок (цена, количество поставляемых изделий). Каждая строка таблицы описывает одну конкретную поставку. Эту таблицу можно трактовать нестрого как задание реляционного отношения. Для строгого определения потребуются следующие понятия.

Типы данных, используемые в реляционной модели

Реляционная модель требует, чтобы типы используемых данных были *простыми*. ***Простые, или атомарные, типы данных*** не обладают внутренней структурой. 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.

Конечно, 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.

Прежде чем говорить о целостности сущностей, опишем использование null-значений в реляционных базах данных.

Null-значения данных

Основное назначение баз данных состоит в том, чтобы хранить и предоставлять информацию о реальном мире. Для представления этой информации в базе данных используются привычные для программистов типы данных – строковые, численные, логические и т. п. Однако в реальном мире часто встречается ситуация, когда данные неизвестны или неполны. Например, место жительства или дата рождения человека могут быть неизвестны (база данных разыскиваемых преступников). Если вместо неизвестного адреса уместно было бы вводить пустую строку, то что вводить вместо неизвестной даты? Ответ – пустую дату – не вполне удовлетворителен, т. к. простейший запрос «выдать список людей в порядке возрастания дат рождения» даст заведомо неправильных ответ.

Для того чтобы обойти проблему неполных или неизвестных данных, в базах данных могут использоваться типы данных, пополненные так называемым ***null-значением***. null-значение – это, собственно, не значение, а некий маркер, показывающий, что значение неизвестно.

Таким образом, в ситуации, когда возможно появление неизвестных или неполных данных, разработчик имеет на выбор два варианта.

Первый вариант состоит в том, чтобы ограничиться использованием обычных типов данных и не использовать null-значения, а вместо неизвестных данных вводить либо нулевые значения, либо значения специального вида – например, договориться, что строка «АДРЕС НЕИЗВЕСТЕН» и есть те данные, которые нужно вводить вместо неизвестного адреса. В любом случае на пользователя (или на разработчика) ложится ответственность за правильную трактовку таких данных. В частности, может потребоваться написание специального программного кода, который в нужных случаях «вылавливал» бы такие данные. Проблемы, возникающие при этом, очевидны – не все данные становятся равноправны, требуется дополнительный программный код, отслеживающий эту неравноправность, в результате чего усложняется разработка и сопровождение приложений.

Второй вариант состоит в использовании null-значений вместо неизвестных данных. При таком подходе есть менее очевид-

ные и более глубокие проблемы. Наиболее бросающейся в глаза проблемой является необходимость использования трехзначной логики при оперировании с данными, которые могут содержать null-значения. В этом случае при неаккуратном формулировании запросов даже самые естественные запросы могут давать неправильные ответы.

Подробное обсуждение проблем использования null-значений выходит за пределы данного учебного пособия, подробное обсуждение проблем, возникающих при использовании null-значений приведено в книге [1].

Практически все реализации современных реляционных СУБД позволяют использовать null-значения, несмотря на их недостаточную теоретическую обоснованность.

Домены

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

Определение 2.1. Домен – это подмножество значений некоторого типа данных имеющих определенный смысл.

Домен характеризуется следующими свойствами:

- домен имеет *уникальное имя* (в пределах базы данных);
- домен определен на некотором *простом* типе данных или на другом домене;
- домен может иметь некоторое *логическое условие*, позволяющее описать подмножество данных, допустимых для данного домена;
- домен несет определенную *смысловую нагрузку*.

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

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

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*, определенную 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. 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. Например, по-видимому, невозможно задать условие на строковый тип данных, задающий домен «Фамилия сотрудника». Ясно, что строки, являющиеся фамилиями, не должны начинаться с цифр, служебных символов, мягкого знака и т. д. Но вот является ли допустимой фамилия, состоящая из бессмысленного набора букв типа «Нпрнекккыыыов»? Очевидно, нет! Трудности такого рода возникают потому, что смысл реальных явлений далеко не всегда можно формально описать. Просто человек интуитивно понимает, что такое фамилия, но никто не может дать такое формальное определение, которое отличало бы фамилии от строк, фамилиями не являющимися. Выход из этой ситуации простой – положиться на разум сотрудника, вводящего фамилии в компьютер.

Определение 2.2. 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.

Реляционное отношение (далее просто отношение), определенное на множестве доменов с именами $D_1, D_2, \ldots, D_n$, – это именованный объект, содержащий две части: ***заголовок*** и ***тело.***

Заголовок $\{A_1: D_1, ..., A_n: D_n\}$ – это множество имен атрибутов или, точнее, пар вида (*имя_атрибута*:*имя_домена*). Тело – это множество ***кортежей*** $\{t_1, t_1, \ldots, t_m\}$, где *i*-й кортеж имеет вид $t_i = \{A_1: v_{i1}, ..., A_n: v_{in}\}$, где v_{ij} – значение *j*-го атрибута A_j в *i*-м кортеже, $v_{ij} \in D_j$.

$$\left\{\begin{array}{l} \{A_1: v_{11},\ A_2: v_{12}, ..., A_n: v_{1n}\} \\ \{A_2: v_{21},\ A_2: v_{22}, ..., A_n: v_{2n}\} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \{A_m: v_{m1},\ A_2: v_{m2}, ..., A_n: v_{mn}\} \end{array}\right\} \qquad (2.1)$$

Все кортежи содержат одно и то же количество *n* элементов (число *n* называется арностью или степенью отношения).

В дальнейшем значение атрибута A_i в кортеже *t* будем обозначать с помощью точечной нотации $t.A_i$.

Свойства отношений:

- в теле отношения нет одинаковых кортежей;
- кортежи не упорядочены сверху вниз;
- атрибуты не упорядочены слева направо.

Подчеркнем, что все значения атрибутов – атомарные, т. е. относятся к простым типам данных в указанном выше смысле. В этом случае говорят, что отношение находится в *первой нормальной форме* (1НФ) или *нормализовано*.

Отношение удобно представлять в виде таблицы, в которой имена столбцов – это атрибуты, а строки представляют кортежи, как показано в табл. 2.2.

Таблица 2.2

Представление реляционного отношения

A_1	A_2	.....	A_n
v_{11}	v_{12}	.....	v_{1n}
v_{21}	v_{22}	.....	v_{2n}
.....	.....	.....	.....
v_{m1}	v_{m2}	.....	v_{mn}

В дальнейшем термины «отношение» и «таблица» будут использоваться как синонимы, хотя, строго говоря, отношение и таблица – это не совсем одно и то же. В отличие от отношения, в таблице столбцы (атрибуты) и строки (кортежи) уже идут в определенном порядке.

Определение 2.3. Реляционная база данных (РБД) – это БД, воспринимаемая пользователем как набор нормализованных отношений разной арности.

Выражение «воспринимаемая пользователем» является решающим: идея реляционной модели применяется к концептуальному (и внешнему) уровню системы, а не к внутреннему уровню.

В дальнейшем в некоторых примерах, чтобы не конкретизировать тело отношения (заполнение таблицы конкретными строками), будем использовать т. н. *схемы отношений. Схема отношения R* – это запись вида $R\{A_1,...,A_n\}$, где R – имя отношения, $A_1,...,A_n$ – имена атрибутов. Примером является следующая схема:

ПОСТАВЩИК-ТОВАР{код_пост, имя_пост, город, код_тов, назв_т, цена_1, кол_во}.

В схеме $R\{A_1,...,A_n\}$ символ R играет роль *переменной отношения*. Аналогично переменным определенного типа в языках программирования переменная R может принимать некоторые значения, т. е. определенные заполнения тела отношения. Использовать понятие «переменная отношения» удобно при обсуждении разного рода теоретических вопросов, когда конкретное тело отношения не играет роли.

Почему сегодня так широко используется реляционная модель данных? Дело в том, что отношения можно рассматривать как математические объекты, а это дает возможность работы с ними (и, следовательно, с РБД) строго формализованными математическими процедурами.

2.2. Целостность базы данных: потенциальные и внешние ключи

В любой момент РБД содержит некоторую определенную конфигурацию данных, и эта конфигурация должна отражать реальную действительность (целостность данных). Следовательно, определение РБД нуждается в расширении, включающем *правила*

целостности данных, назначение которых в том, чтобы информировать СУБД о разного рода ограничениях реального мира.

Есть правила целостности, которые относятся к конкретной БД, например:

– количество поставляемых деталей > 0;

– код поставщика имеет формат Пdd , где *d* – цифра;

– атрибут *город* принимает значения из определенного списка.

Это специфические для конкретных отношений правила целостности, их выполнение – задача разработчика БД. Однако есть два общих особых правила целостности, которые должны выполняться для *любой* РБД. Эти правила связаны с понятием *потенциальных* и *внешних ключей*.

2.2.1. Потенциальные ключи

Пусть $R\{A_1,...,A_n\}$ – схема некоторого отношения. Пусть $K \subseteq \{A_1,...,A_n\}$ – некоторое подмножество атрибутов.

Определение 2.4. Подмножество *K* называется потенциальным ключом, если оно удовлетворяет следующим свойствам:

1. Уникальность ключа – в любой момент в R нет двух различных кортежей с одинаковым значением K;
2. Неизбыточность ключа – $\forall K' \subset K$ не выполняется свойство уникальности.

Пример. В отношении *ПОСТАВЩИК_ТОВАР{код_пост, имя_пост, город, код_тов, название, цена_1, кол_во}* потенциальный ключ $K = \{код\_пост, код\_товара\}$.

Замечание. Возможны отношения, в которых единственным потенциальным ключом будет комбинация *всех* атрибутов, например, это справедливо для отношения со схемой *ПОСТАВЩИК-ТОВАР_1(код_пост, код_тов).*

Для чего нужны потенциальные ключи? Они обеспечивают основной механизм адресации на уровне кортежей. Иными словами, единственный гарантируемый системой способ точно указать на какой-либо кортеж отношения – это указать значение некоторого потенциального ключа.

Если в реляционном отношении есть несколько потенциальных ключей, то один из них указывается в качестве *первичного* ключа, остальные считаются *альтернативными* ключами.

Первое правило целостности данных (целостность сущностей) – *в любом отношении реляционной базы данных должен быть указан первичный ключ.*

В принципе при создании таблицы СУБД должна запрашивать у разработчика первичный ключ и при работе контролировать состояние таблицы после каждой ее модификации, не допуская наличия разных кортежей с одинаковым значением первичного ключа.

Далее в схеме отношения атрибуты, входящие в первичный ключ, будут выделяться подчеркиванием, например

СТУДЕНТ-ОЦЕНКА{*номер_студ_билета, фамилия, группа, дисциплина, оценка*}.

Второе правило, которое называется *правилом целостности по ссылкам*, является более сложным. Для этого потребуется ввести понятие внешних ключей.

2.2.2. Внешние ключи

Понятие внешних ключей поясним на примере реляционной базы данных со следующей схемой:

ТОВАР(код_товара, название, фирма_производитель) – справочник деталей;

ПОСТАВЩИК(код_поставщика, имя_поставщика, город) – справочник поставщиков;

ПОСТАВЩИК-ТОВАР(код_пост, код_тов, кол_во) – данные о поставках.

Рассмотрим отношение *ПОСТАВЩИК-ТОВАР*. Конкретное значение атрибута *код_пост* допустимо для отношения *ПОСТАВЩИК-ТОВАР* лишь в том случае, если такое же значение существует в качестве первичного ключа *K={код_поставщика}* в отношении *ПОСТАВЩИК*. Другими словами, не имеет смысла включать в *ПОСТАВЩИК-ТОВАР* поставку для поставщика *П07*, если в справочнике поставщиков *ПОСТАВЩИК отсутствует* поставщик с кодом *П07*. Аналогичная ситуация для деталей.

Определение 2.5. Пусть $R1$ – отношение. Тогда внешний ключ $L1$ в $R1$ – это подмножество атрибутов отношения $R1$, такое что

- существует отношение $R2$ с первичным ключом $L2$;
- каждое значение $L1$ в текущем значении $R1$ совпадает со значением $L2$ некоторого кортежа в текущем значении $R2$ (обратное не требуется!).

Иначе говоря, внешний ключ – это атрибут (или подмножество атрибутов), чьи значения совпадают с имеющимися значениями первичного ключа другого отношения. В приведенном выше примере атрибут *код_пост* в отношении *ПОСТАВЩИК-ТОВАР* является внешним ключом, связывающим это отношение с отношением *ПОСТАВЩИК,* т. к. в *ПОСТАВЩИК* атрибут *код_поставщика,* имеющий тот же смысл, что и *код_пост,* является первичным ключом.

Подобное взаимоотношение между таблицами называется *связью (relationship)*. Связь между двумя таблицами устанавливается путем присвоения значений внешнего ключа одной таблицы значениям первичного ключа другой.

Как правило, внешний ключ отношения *R* является частью потенциального ключа этого же отношения. Однако это не обязательно, например, в БД со схемой

ОТДЕЛ(номер, название, фонд_зарплаты),
СЛУЖАЩИЙ(таб_номер, ФИО, номер_отдела, зарплата, должность)

в отношении *СЛУЖАЩИЙ* атрибут *номер_отдела* является внешним ключом (относительно *ОТДЕЛ*), но он не входит ни в один потенциальный ключ этого отношения.

Второе правило целостности данных (*целостность по ссылкам*) – *БД не содержит несогласованных значений внешних ключей*.

Для отображения связей между отношениями (ссылок по внешним ключам) удобно использовать диаграммы следующего вида (рис. 2.1):

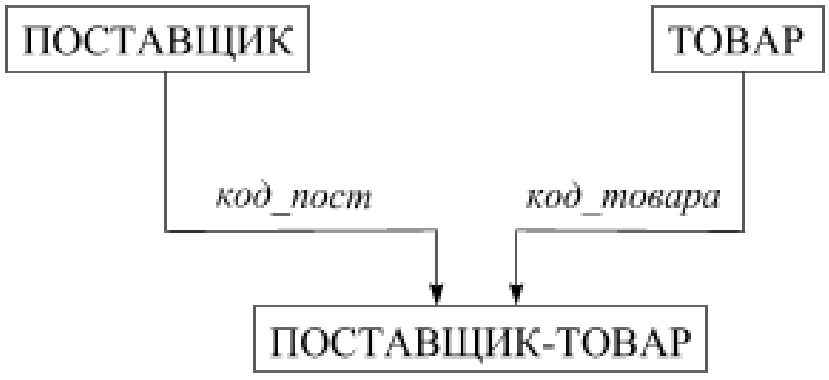


Рис. 2.1. Связь между таблицами по внешним ключам

При наличии связи $R1 \xrightarrow{L} R2$ по внешнему ключу *L* отношение *R*1 называется *главным* (master-table), а отношение *R*2 –

подчиненным (detail-table) в данной связи. Так, на рис. 2.1 отношения *ПОСТАВЩИК* и *ТОВАР* – главные по отношению к подчиненному отношению *ПОСТАВЩИК-ТОВАР*.

2.2.3. Как обеспечивается ссылочная целостность

Второе правило целостности выражено исключительно в терминах *состояний* БД. Любое состояние БД, не удовлетворяющее этому правилу, некорректно. Одна из возможностей избежать некорректности – это запретить любые операции, приводящие к этому состоянию. Однако в некоторых случаях предпочтительнее допустить такую операцию, но при необходимости выполнить некоторые *компенсирующие* операции. Например, если пользователь удаляет из отношения *ПОСТАВЩИК* поставщика с кодом ’П01’ система сама может удалить все поставки этого поставщика из отношения *ПОСТАВЩИК-ТОВАР* (т. н. *каскадное удаление*). Следовательно, у разработчика БД должна быть возможность определить, какие операции должны быть запрещены, а какие разрешены, нужны ли для разрешенных операций компенсирующие, и если да, то какие именно (язык SQL позволяет это делать).

После назначения внешнего ключа СУБД имеет возможность автоматически отслеживать вопросы «ненарушения» связей между отношениями, а именно:

- если пользователь попытается вставить в подчиненную таблицу запись, для внешнего ключа которой не существует соответствия в главной таблице (например, там нет еще записи с таким первичным ключом), СУБД сгенерирует ошибку;
- если пользователь попытается удалить из главной таблицы запись, на первичный ключ которой имеется хотя бы одна ссылка из подчиненной таблицы, СУБД также сгенерирует ошибку;
- если пользователь попытается изменить первичный ключ записи главной таблицы, на которую имеется хотя бы одна ссылка из подчиненной таблицы, СУБД также сгенерирует ошибку.

Замечание. Существуют два подхода к удалению и изменению записей в главной таблице:

- запретить удаление всех записей, а также изменение первичных ключей главной таблицы, на которые имеются ссылки подчиненной таблицы;

- распространить всякие изменения в первичном ключе главной таблицы на подчиненную таблицу, а именно:
  - если в главной таблице удалена запись, то в подчиненной таблице должны быть удалены все записи, ссылающиеся на удаляемую;
  - если в главной таблице изменен первичный ключ записи, то в подчиненной таблице должны быть изменены (автоматически) все внешние ключи записей, ссылающихся на изменяемую.

2.3. Средства манипулирования реляционными данными

Третий компонент реляционной модели (помимо понятия отношений и целостности данных) включает в себя набор операторов, которые дают возможность разработчику реляционной БД генерировать новые отношения из старых (базовых) для решения прикладных задач. Существуют два подхода к построению таких операторов – *реляционная алгебра* и *реляционное исчисление.*

В реализациях конкретных реляционных СУБД сейчас не используется в чистом виде ни реляционная алгебра, ни реляционное исчисление. Фактическим стандартом доступа к реляционным данным стал язык SQL (Structured Query Language). Язык SQL представляет собой смесь операторов реляционной алгебры и выражений реляционного исчисления, использующий синтаксис, близкий к фразам английского языка и расширенный дополнительными возможностями, отсутствующими в реляционной алгебре и реляционном исчислении. Вообще, язык доступа к данным называется ***реляционно полным***, если он по выразительной силе не уступает реляционной алгебре (или, что то же самое, реляционному исчислению), т. е. любой оператор реляционной алгебры может быть выражен средствами этого языка. Именно таким и является язык SQL.

2.3.1. Реляционная алгебра Кодда

Реляционная алгебра представляет собой набор операторов, использующих отношения в качестве аргументов и возвращающих отношения в качестве результата. Таким образом, реляцион-

ный оператор выглядит как функция с отношениями в качестве аргументов:

$$R = f(R_1, R_1, \ldots, R_n).$$

Реляционная алгебра является замкнутой, т. к. в качестве аргументов в реляционные операторы можно подставлять другие реляционные операторы, подходящие по типу:

$$R = f(f_1(R_{11}, \ldots, R_{1k_1}), f_2(R_{21}, \ldots, R_{2k_2}), \ldots).$$

Таким образом, в реляционных выражениях можно использовать вложенные выражения сколь угодно сложной структуры.

Каждое отношение обязано иметь уникальное имя в пределах базы данных. Имя отношения, полученного в результате выполнения реляционной операции, определяется в левой части равенства. Однако можно не требовать наличия имен от отношений, полученных в результате реляционных выражений, если эти отношения подставляются в качестве аргументов в другие реляционные выражения. Такие отношения будем называть *неименованными отношениями*. Неименованные отношения реально не существуют в базе данных, а только вычисляются в момент вычисления значения реляционного оператора.

Существует много подходов к определению наборов операторов и способов их интерпретации, но в принципе все они являются более или менее равносильными. Здесь будет рассмотрен классический подход, предложенный Кристофером Коддом. В этом варианте множество операторов состоит из восьми операций, составляющих две группы по четыре оператора в каждой:

- Традиционные операторы над множествами: *объединение, пересечение, вычитание* и *декартово произведение* (все они модифицированы с учетом того, что их операндами являются отношения, а не произвольные множества).
- Специальные реляционные операторы: *выборка, проекция, соединение и деление*.

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

Операция присваивания имеет вид

R =<выражение реляционной алгебры>,

где R – переменная отношения.

Операция переименования атрибута – это выражение вида

$$R \text{ RENAME } A_1, A_2, \ldots \text{ AS } B_1, B_2, \ldots,$$

где R – переменная отношения, $A_1, A_2, \ldots$ – имена атрибутов отношения R, $B_1, B_2, \ldots$ – новые имена. Это выражение приводит к получению отношения с тем же заголовком и телом, что и отношение, которое является текущим значением переменной R, за исключением того, что в нем атрибуты A_i называются теперь B_i.

Два отношения называются *совместимыми по типу*, если они имеют одинаковые заголовки. Некоторые отношения не являются совместимыми по типу, но становятся таковыми после некоторого переименования атрибутов.

Теоретико-множественные операторы

Объединение. Объединением двух совместимых по типу реляционных отношений A и B (A UNION B) называется отношение C с тем же заголовком, что у A и B, и с телом, состоящим из множества всех кортежей, принадлежащим A или B или обоим вместе:

$$t \in A \ \ UNION \ \ B \Leftrightarrow t \in A \ \ OR \ \ t \in B.$$

Оператор объединения поясняется на рис. 2.2 (а).

Пересечение. Пересечением двух совместимых по типу отношений R и S (синтаксис R INTERSECT S) называется отношение с тем же заголовком, что у R и S, и с телом, состоящим из множества всех кортежей, принадлежащим R и S:

$$t \in A \ \ INTERSECT \ \ B \Leftrightarrow t \in A \ \ AND \ \ t \in B.$$

Оператор пересечения поясняется на рис. 2.2 (б).

Вычитание. Вычитанием двух совместимых по типу отношений A и B (A MINUS B) называется отношение с тем же заголовком, что у A и B, и с телом, состоящим из множества всех кортежей, принадлежащим A и не принадлежащим B:

$$t \in A \ \ MINUS \ \ B \Leftrightarrow t \in A \ \ AND \ \ t \notin B.$$

Оператор вычитания поясняется на рис. 2.2 (в).

Декартово произведение. Пусть имеются два отношения $R\{A_1, A_2, \ldots, A_m\}$ и $S\{B_1, B_2, \ldots, B_n\}$. Тогда результатом операции произведения *R* TIMES *S* является отношение $T\{A_1, \ldots, A_m, B_1, \ldots, B_n\}$, тело которого является множеством кортежей вида $\{r_{A_1}, \ldots, r_{A_m}, r_{B_1}, \ldots, r_{B_n}\}$ таких, что $\{r_{A_1}, \ldots, r_{A_m}\}$ входит в тело *R*, а $\{r_{B_1}, \ldots, r_{B_n}\}$ входит в тело отношения *S*.

Оператор декартового произведения поясняется на рис. 2.2 (г).

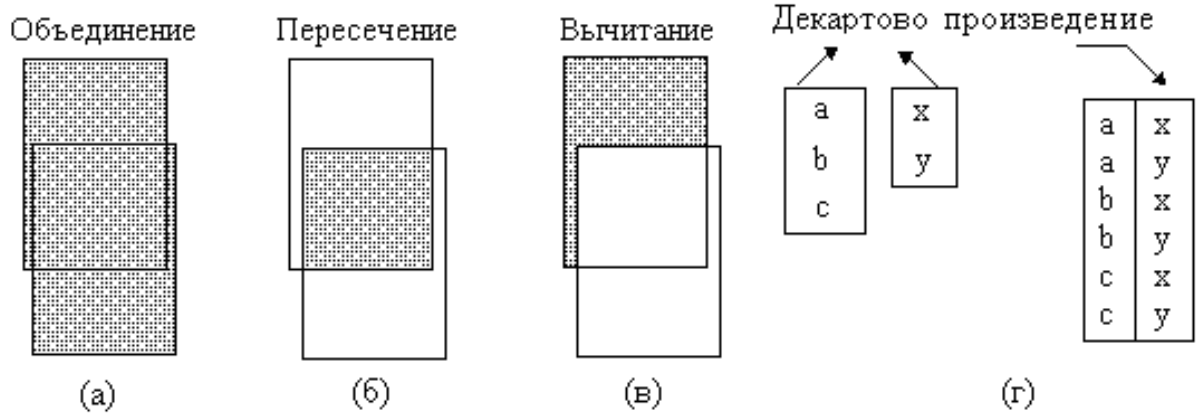


Рис. 2.2. Иллюстрация теоретико-множественных операций

Здесь может возникнуть проблема – как получить корректно сформированный заголовок отношения-результата? Поскольку заголовок результирующего отношения является сцеплением заголовков отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

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

Следует заметить, что операция взятия декартова произведения не является слишком осмысленной на практике. Во-первых, мощность тела ее результата очень велика даже при допустимых мощностях операндов, а, во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как будет

показано далее, основной смысл включения операции расширенного декартова произведения в состав реляционной алгебры Кодда состоит в том, что на ее основе определяется действительно полезная операция соединения.

Специальные операторы

Выборка (θ-выборка). Пусть $\theta \in \{=, >, <, \geq, \leq, \neq\}$ – символ операции сравнения, R – отношение. Тогда θ-выборка (R WHERE $X\theta Y$) – это отношение с тем же заголовком, что и R, содержащее кортежи t из R, для которых условие $X\theta Y$ истинно. Здесь X – атрибут, а Y – атрибут, определенный на том же домене, что и X, или литерал (т. е. символьная строка или число).

Пример. Рассмотрим отношение *ПОСТАВЩИК-ТОВАР* (табл. 2.1). Тогда выражение

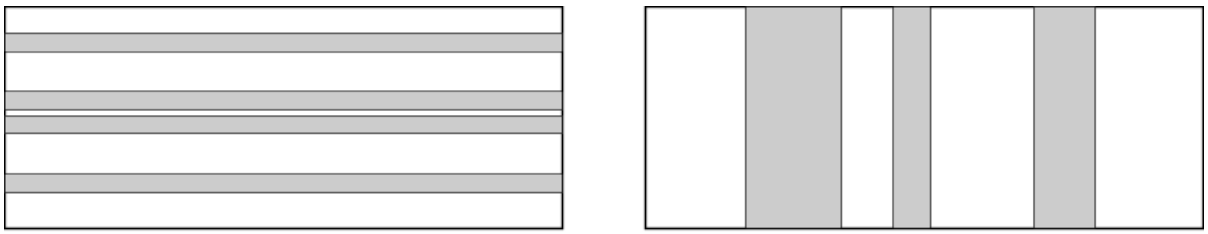
ПОСТАВЩИК-ТОВАР WHERE *назв_т* = ‘Монитор’ определяет отношение с телом вида

код_п	*имя_п*	*гор*	*код_т*	*назв_т*	*цена_1*	*кол_во*
П01	Скан	Ярославль	Т14	монитор	7 500	10
П11	Альфа	Москва	Т14	монитор	7 200	3

Обычно определение выборки расширяется до формы, в которой условие в выражении WHERE будет содержать произвольное число логических сочетаний простых условий, например,

ПОСТАВЩИК-ТОВАР WHERE (*назв_т* = ‘монитор’)
AND (*гор* = ‘Москва’ OR *гор* = ‘Воронеж’).

На интуитивном уровне оператор выборки лучше всего представлять как взятие некоторой «горизонтальной» вырезки из отношения-операнда (выборки некоторых строк из таблицы), как показано на рис. 2.3 (а).



(а) Выборка (б) Проекция

Рис. 2.3. Операции выборки и проекции

Проекция. Проекция отношения *R* на атрибуты $A, B, \ldots, C$ (синтаксис $R[A, B, \ldots, C]$) – это отношение с заголовком $A, B, \ldots, C$ и телом, состоящим из кортежей $\{A : a, B : b, \ldots, C : c\}$, таких, для которых в отношении *R* значение атрибута *A* равно *a*, атрибута *B* равно *b*,..., атрибута *C* равно *c*. Таким образом, проекция – это подмножество кортежей, получаемое исключением тех атрибутов, которые не указаны в списке атрибутов, и последующим исключением дублирующих подкортежей. Тем самым при выполнении операции проекции выделяется «вертикальная» вырезка отношения-операнда (рис. 2.3 (б)).

Пример. Рассмотрим снова отношение *ПОСТАВЩИК–ТОВАР* (см. табл. 2.1). Выражение

ПОСТАВЩИК-ТОВАР[назв_ т, кол_во]

формирует отношение со следующими кортежами:

Назв_т	*кол_во*
Монитор	10
Принтер	16
Монитор	3
Процессор	3

Соединение. Оператор соединения (называемый также *соединением по условию, или θ-соединением*) требует наличия двух операндов – соединяемых отношений и третьего операнда – простого условия. Пусть соединяются отношения *R* и *S*. Тогда, по определению, результатом операции соединения (*R* JOIN *S*) WHERE usl совместимых по взятию декартова произведения отношений *R* и *S* является отношение, получаемое путем выполнения операции выборки по условию usl декартова произведения отношений *R* и *S*:

(*R* JOIN *S*) WHERE usl ≡ (*R* TIMES *S*) WHERE usl.

Хотя операция соединения в приведенной интерпретации не является примитивной (поскольку определяется с использованием операций декартова произведения и проекции), в силу особой практической важности она включается в базовый набор операций реляционной алгебры Кодда. Заметим также, что в практи-

ческих реализациях соединение обычно не выполняется именно как выборка декартова произведения. Имеются более эффективные алгоритмы, гарантирующие получение такого же результата.

Важным частным случаем операции соединения по условию является *естественное соединение.* Операция естественного соединения применяется к паре отношений $R\{A,B\}$ и $S\{B,C\}$, обладающих (возможно, составным) общим атрибутом B (т. е. атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть ABC обозначает объединение заголовков отношений A и B. Тогда естественное соединение отношений R и S $(R\ NATURAL\ JOIN\ S)$ – это спроецированный на ABC результат соединения R и S по условию $R.B = S.B$[1]. Хотя операция естественного соединения выражается через операции переименования, соединения по условию и проекции, для нее обычно используется сокращенная форма, называемая NATURAL JOIN.

Пример. Пусть отношения *ПОСТАВЩИК* и *ПОСТАВЩИК-ТОВАР* имеют следующее заполнение:

ПОСТАВЩИК

код_п	*имя_п*	*гор*
П01	Скан	Ярославль
П02	Альфа	Москва
П03	Тензор	Ярославль

ПОСТАВЩИК-ТОВАР

код_п	*код_т*	*цена_1*	*кол_во*
П01	Т06	23	10
П01	Т04	234	5
П03	Т06	120	6

Выражение *ПОСТАВЩИК* NATURAL JOIN *ПОСТАВЩИК–ТОВАР* создает новое отношение со следующими кортежами:

код_п	*имя_п*	*гор*	*код_т*	*цена1*	*кол_во*
П01	Скан	Ярославль	Т06	123	10
П01	Скан	Ярославль	Т04	234	5
П03	Тензор	Ярославль	Т06	120	6

Замечание. В синтаксисе естественного соединения не указываются, по каким атрибутам производится соединение. Естественное соединение производится *по всем* одинаковым атрибутам.

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

Замечание. Естественное соединение эквивалентно следующей последовательности реляционных операций:

- переименовать одинаковые атрибуты в отношениях,
- выполнить декартово произведение отношений,
- выполнить выборку по совпадающим значениям атрибутов, имевших одинаковые имена,
- выполнить проекцию, удалив повторяющиеся атрибуты,
- переименовать атрибуты, вернув им первоначальные имена.

Операция деления отношений. Эта операция наименее очевидна из всех операций реляционной алгебры Кодда и поэтому нуждается в более подробном объяснении. Пусть заданы два отношения – $R\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$ (делимое) и $S\{B_1, B_2, \ldots, B_m\}$ (делитель). Будем считать, что атрибут B_i отношения R и атрибут B_i отношения S $(i = 1, \ldots, m)$ не только обладают одним и тем же именем, но и определены на одном и том же домене.

По определению, результатом деления R на S (R DIVIDE BY S) является отношение $T\{A_1, A_2, \ldots, A_n\}$ (частное), которое состоит из кортежей $t = \{a_1, a_2, \ldots, a_n\}$ таких, что для *всех* кортежей $\{b_1, b_2, \ldots, b_m\} \in S$ в отношении R найдется кортеж $\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\}$. Другими словами, тело декартова произведения T TIMES S содержится в теле отношения R. Поясним это определение на простейшем примере.

Пример. Рассмотрим отношения следующего вида:

R

A	*B*
a1	b1
a1	b2
a2	b1

S

B
b1
b2

Тогда команда $T = R$ DIVIDE BY S сформирует отношение T, тело которого содержит один кортеж

A
a1

Ситуацию, в которой удобно использовать операцию реляционного деления, проиллюстрируем на более содержательном примере. Типичные запросы, реализуемые с помощью операции деления, обычно в своей формулировке имеют слово «все» – «Какие поставщики поставляют *все* детали?», «Какие сотрудники работают во *всех* проектах?» и т. п. Пусть в БД некоторой организации поддерживаются два отношения: *ПРОЕКТЫ*{*проект,*

объем_финанс}, представляющее справочник всех проектов, выполняемых в организации, и *ПРОЕКТЫ-СОТРУДНИКИ{проект, сотр_имя*, *отд_номер* }, где для каждого проекта указаны все задействованные в этом проекте сотрудники и отделы, в которых они работают. Требуется составить список всех тех сотрудников, которые работают ***во всех*** проектах. Требуемый список может быть получен в виде отношения *СОТРУД-ВСЕ-ПРОЕКТЫ*{*имя_сотруд, отдел*} следующим образом:

СОТРУД-ВСЕ-ПРОЕКТЫ = ПРОЕКТ-СОТРУДНИК
DIVIDE BY *ПРОЕКТЫ[проект]*.

Этот результат поясняется ниже на конкретных значениях переменных отношений *ПРОЕКТЫ* и *ПРОЕКТ-СОТРУДНИК*.

Отметим, что операция реляционного деления не является примитивной – она выражается через операции декартова произведения, взятия разности и проекции.

Основная цель создания реляционной алгебры – возможность формирования запросов к базе данных на формальном языке, операторами которого являются введенные операторы алгебры Кодда.

ПРОЕКТ-СОТРУДНИК-ОТДЕЛ

проект	***сотр_имя***	***отд_номер***
Пр1	Меньшиков О.Е.	1
Пр1	Домогаров А.Ю.	2
Пр1	Безруков С.В.	2
Пр2	Меньшиков О.Е.	1
Пр2	Домогаров А.Ю.	2
Пр2	Безруков С.В.	2
Пр3	Домогаров А.Ю.	2
Пр3	Боярская Е.М.	2
Пр3	Безруков С.В.	2

ПРОЕКТЫ

проект	***объем_финансирования***
Пр1	***100000***
Пр2	***234000***
Пр3	150000

СОТРУД-ВСЕ-ПРОЕКТЫ

сотр_имя	***отд_номер***
Домогаров А.Ю.	2
Безруков С.В.	2

2.3.2. Реляционное исчисление

Здесь мы поясним основной принцип, лежащий в основе реляционного исчисления. Более подробно с реализациями этого

подхода к работе с реляционными отношениями можно ознакомиться в работах [1–3].

Как уже было сказано, в реляционной модели определяются два базовых механизма манипулирования данными:

- основанная на теории множеств реляционная алгебра (алгебра Кодда),
- основанное на математической логике реляционное исчисление.

Так же, как и выражения реляционной алгебры, формулы реляционного исчисления определяются над отношениями реляционных баз данных, и результатом вычисления также является отношение. Эти механизмы манипулирования данными различаются уровнем процедурности:

- запрос, представленный на языке реляционной алгебры, может быть вычислен на основе вычисления элементарных алгебраических операций с учетом их старшинства и возможных скобок;
- формула реляционного исчисления только формулирует (декларирует) *условия*, которым должны удовлетворять кортежи результирующего отношения.

Поэтому языки реляционного исчисления являются более непроцедурными или декларативными.

Пример. Пусть даны два отношения:

СОТРУДНИКИ {*сотр_номер, сотр_имя, сотр_зарпл, отд_номер*},

ОТДЕЛЫ{*отд_номер, отд_кол, отд_нач*}.

Здесь атрибут *отд_кол* означает количество сотрудников в отделе, а *отд_нач* – фамилию начальника отдела. Мы хотим узнать имена и номера сотрудников, являющихся начальниками отделов с количеством работников более 10. Выполнение этого запроса средствами реляционной алгебры распадается на четко определенную последовательность шагов:

1. Выполнить соединение отношений СОТРУДНИКИ и ОТДЕЛЫ по условию *сотр_номер = отдел_нач:*

C1 = СОТРУДНИКИ JOIN ОТДЕЛЫ WHERE сотр_номер = отд_нач.

2. Из полученного отношения произвести выборку по условию *отд_кол* > 10:

C2 = C1 WHERE отд_кол > 10

3. Спроецировать результаты предыдущей операции на атрибуты *сотр_имя, сотр_номер:*

C3 = C2 [сотр_имя, сотр_номер].

Заметим, что порядок выполнения шагов может повлиять на эффективность выполнения запроса. Так, время выполнения приведенного выше запроса можно сократить, если поменять местами этапы 1 и 2. В этом случае сначала из отношения СОТРУДНИКИ будет сделана выборка всех кортежей со значением атрибута отдел_кол > 10, а затем выполнено соединение результирующего отношения с отношением ОТДЕЛЫ. Машинное время экономится за счет того, что в операции соединения участвуют меньшие отношения.

На языке реляционного исчисления данный запрос может быть записан как:

```
Выдать сотр_имя и сотр_ном для СОТРУДНИКИ таких, что
существует ОТДЕЛ с таким же, что и сотр_ном значением
отд_нач и значением отд_кол большим 10.
```

Здесь мы указываем лишь характеристики результирующего отношения, но не говорим о способе его формирования. СУБД сама должна решить, какие операции и в каком порядке надо выполнить над отношениями СОТРУДНИКИ и ОТДЕЛЫ. Задача оптимизации выполнения запроса в этом случае также ложится на СУБД.

2.3.3. Ограниченность реляционной алгебры

Несмотря на мощь языка реляционной алгебры, имеются такие типы запросов, которые принципиально нельзя выразить только при помощи операторов реляционной алгебры. Это вовсе не означает, что ответы на эти запросы нельзя получить вообще. Просто для получения ответов на подобные запросы приходится применять процедурные расширения реляционных языков. Приведем примеры двух задач, неразрешимых в реляционной алгебре [2].

Пример. Пусть имеется отношение *ХИМ_СОСТАВ_ВЕЩЕСТВ* с набором атрибутов (*Наименование вещества, Водород, Гелий, …, 105-й элемент*). Значением атрибута «*Вещество*» являются наименования химических веществ, значениями остальных атрибутов – процентный состав соответствующих элементов в этом веществе. Такое отношение могло бы иметь, к примеру, следующий вид:

ХИМ_СОСТАВ_ВЕЩЕСТВ

Наименование вещества	Водород	Гелий	…	105-й элемент
Дезоксирибонуклеиновая кислота	5	3	…	0.01
Бензин	50	0	…	0
…	…	…	…	…

Рассмотрим запрос «Найти все химические элементы, содержание которых в каком-либо из веществ превышает заданный процент (скажем, 90)».

С алгоритмической точки зрения этот запрос выполняется элементарно – просматриваются все столбцы таблицы; если в столбце присутствует хотя бы одно значение, большее 90, то запоминается заголовок этого столбца. Набор наименований запомненных столбцов и является ответом на запрос.

Формально невозможно выразить этот запрос в рамках реляционной алгебры, т. к. ответом на этот запрос должен быть *список атрибутов* отношений, удовлетворяющих определенному условию. В реляционной алгебре нет операторов, манипулирующих с наименованиями атрибутов.

На самом деле, этот пример показывает, что таблица плохо нормализована (нормализация отношений рассматривается в гл. 4 и 5). В таблице есть набор однотипных атрибутов («Водород», «Гелий» и т. д. в количестве 105 столбцов).

Правильнее разбить это отношение на три различных отношения:

ВЕЩЕСТВО{*ном_вещества, вещество*},
ЭЛЕМЕНТЫ{*ном_элемента, элемент*},
ХИМ_СОСТАВ_ВЕЩЕСТВ{*ном_вещества, ном_элемента, процент*}

ХИМ_СОСТАВ_ВЕЩЕСТВ

ном_вещества	*ном_элемента*	*процент*
1	1	5
1	2	3
…	…	…

ВЕЩЕСТВО

ном_вещества	*вещество*
1	Дезоксирибонуклеиновая кислота
2	Бензин
…	……

ЭЛЕМЕНТЫ

ном_элемента	*элемент*
1	Водород
2	Гелий
…	…

Для отношений, нормализованных таким образом, исходный запрос реализуется следующей последовательностью операторов:

R1 = ХИМИЧЕСКИЙ_СОСТАВ_ВЕЩЕСТВ WHERE *процент*>90. (Выборка из отношения).
R2 = R1[*ном_элемента*]. (Проекция отношения).
R3 = R2 NATURAL JOIN ЭЛЕМЕНТЫ. (Естественное соединение)
ОТВЕТ = R3[*элемент*]. (Проекция таблицы).

Пример. Одной из задач, связанных с представлением табличных данных, является построение так называемых кросс-таблиц. Пусть имеется отношение с тремя атрибутами и потенциальным ключом, включающим первые два атрибута. Примером такого отношения могут быть данные с объемами продаж различных товаров за некоторые промежутки времени (табл. 2.3).

Требуется представить эти данные в виде таблицы, по строкам которой идут наименования товаров, по столбцам – месяцы, а в ячейках содержатся объемы продаж. Это и будет кросс-таблицей (табл. 2.4).

Таблица 2.3

ДАННЫЕ О ПРОДАЖАХ

товар	*месяц*	*кол-во*
Компьютеры	Январь	100
Принтеры	Январь	200
Сканеры	Январь	300
Компьютеры	Февраль	150
Принтеры	Февраль	250
Сканеры	Февраль	350
…	…	…

Таблица 2.4

КРОСС-ТАБЛИЦА

товар	*январь*	*февраль*	…
Компьютеры	100	150	…
Принтеры	200	250	…
Сканеры	300	350	…

Построение кросс-таблицы средствами реляционной алгебры невозможно, т. к. для этого требуется превратить данные в ячейках таблицы в наименования новых столбцов таблицы.

Контрольные вопросы

1. Дайте определение каждому из следующих понятий в контексте реляционной модели данных:

а) отношение; б) атрибут; в) домен;

г) кортеж; д) заголовок и тело.

2. Укажите различия между потенциальными ключами и первичным ключом отношения. Что означает понятие «внешний ключ»? Как внешние ключи отношений связаны с потенциальными ключами? Приведите примеры, иллюстрирующие ваши ответы.

3. Дайте определение двух основных правил целостности реляционной модели и расскажите, почему необходимо их использовать.

4. Дайте определение пяти основным операциям реляционной алгебры. Определите оставшиеся три операции реляционной алгебры на основе этих пяти операций.

Практическое задание

Перечисленные ниже таблицы образуют часть базы данных реляционной СУБД

Hotel(*hotel_no, name, address*)

Room(*room_no, hotel_no, type, price*)

Booking(*hotel_no, guest_no, date_from, date_to, room_no*)

Guest(*guest_no, name, address*)

Здесь таблица Hotel содержит сведения о гостинице, причем атрибут *hotel_no* является ее первичным ключом. Таблица Room содержит данные о номерах всех гостиниц, а комбинация атрибутов (*hotel_no, room_no*) образует ее первичный ключ. Таблица Booking содержит сведения о бронировании гостиничных номеров: ее первичным ключом является комбинация атрибутов (*hotel_no, guest_no, date_from*). Наконец, таблица Guest содержит сведения о постояльцах гостиниц, и ее первичным ключом является атрибут *guest_no*.

Напишите выражения реляционной алгебры, позволяющие выполнить следующие запросы:

1) перечислить все гостиницы;

2) перечислить все однокомнатные гостиничные номера стоимостью менее 75 € за сутки;

3) перечислить имена и адреса всех постояльцев;

4) составить список стоимости и типов всех гостиничных номеров в гостинице «Grosvenor Hotel»;

5) перечислить всех постояльцев гостиницы «King Hotel»;

6) привести сведения обо всех номерах гостиницы «King Hotel», включая имена постояльцев, снимающих тот или иной номер;

Объясните, как правила целостности сущностей и ссылочной целостности могут быть применены к этим отношениям.

3. Язык SQL

3.1. Общие сведения

Мы уже упоминали, что основой взаимодействия с реляционными базами данных является язык SQL. Его появление и развитие связано с деятельностью по практическому воплощению идей реляционной модели в коммерческие продукты.

Этапы развития языка SQL закрепляются в стандартах, которые не только фиксируют достигнутый общеприемлемый уровень, но и обозначают направление дальнейшего развития. Важную роль в этой деятельности сыграл стандарт SQL92 (SQL2), совместимость с которым до сих пор является важной характеристикой систем управления базами данных.

Результатом длительной дискуссии об объектно ориентированных базах данных явился стандарт SQL:1999 (SQL3), в который были введены т. н. объектно ориентированные расширения. Претерпела изменения и структура стандарта: он был разделен на несколько книг, каждая из которых посвящена отдельной теме. Кроме того, доступ к официальной редакции стандарта был переведен на коммерческую основу: за умеренную плату (несколько десятков долларов) можно получить электронную версию каждой книги в pdf-формате.

Дальнейшее развитие стандарт языка SQL получил в редакциях SQL:2003 и SQL:2008, в целом сохранивших структуру редакции SQL:1999.

Все редакции стандарта ISO языка SQL имеют одинаковое кодовое обозначение IEC 9075. Вот перечень книг (секций), входящих в SQL:2008:

ISO/IEC 9075-1:2008 Framework (SQL/Framework)

ISO/IEC 9075-2:2008 Foundation (SQL/Foundation)

ISO/IEC 9075-3:2008 Call-Level Interface (SQL/CLI)

ISO/IEC 9075-4:2008 Persistent Stored Modules (SQL/PSM)

ISO/IEC 9075-9:2008 Management of External Data (SQL/MED)

ISO/IEC 9075-10:2008 Object Language Bindings (SQL/OLB)

ISO/IEC 9075-11:2008 Information and Definition Schemas (SQL/Schemata)

ISO/IEC 9075-13:2008 SQL Routines and Types Using the Java TM Programming Language (SQL/JRT)

ISO/IEC 9075-14:2008 XML-Related Specifications (SQL/XML)

Основной и наиболее объемной из них является вторая – SQL/Foundation (Основания).

Несмотря на то что стандарты обозначают некоторое общее понимание того, каким должен быть язык взаимодействия с базой данных, различные производители реализуют его в своих программных продуктах (СУБД) по-разному. Связано это с тем, что для расширения функциональных возможностей и повышения эффективности разработчики конкретной СУБД добавляют к стандартному языку SQL дополнительные команды и функции, исходя из собственного понимания их необходимости, а также отзывов сообщества пользователей этой СУБД, сохраняя при этом некоторые особенности предыдущих версий. Поскольку сферы интересов пользователей различных СУБД отличаются друг от друга, различаются и создаваемые расширения. Можно сказать, что каждая реализация языка SQL представляет собой расширение некоторого подмножества стандарта. Впоследствии разработчики стандарта пытаются обобщить вводимые расширения и ввести их в стандарт наиболее логичным способом, причем этот способ может заметно отличаться от имеющихся реализаций. Поэтому для практической работы с конкретной СУБД, помимо стандарта, необходимо знать и отличительные особенности данной СУБД. Интересной в этой связи является позиция разработчиков СУБД PostgreSQL, заявляющих, что их система полностью соответствует текущей редакции стандарта.

Хотя язык SQL первоначально разрабатывался для поддержки реляционной модели данных, реализуемая им модель несколько отличается от реляционной, что явилось результатом практического использования реляционных баз данных. Из наиболее заметных отличий можно отметить отсутствие требования разли-

чия всех строк в таблицах базы данных (и, следовательно, обязательного наличия первичного ключа), а также возможность наличия т. н. пустых (null) значений в клетках таблицы, обрабатываемых с использованием трехзначной логики.

Основу языка SQL составляют команды (операторы), каждая из которых извлекает информацию из базы данных или производит в ней какие-либо изменения. Среди основных команд SQL можно выделить следующие группы:

команды определения данных (DDL – Data Definition Language): CREATE, DROP, ALTER;

команды манипулирования данными (DML – Data Manipulation Language): SELECT, INSERT, DELETE, UPDATE;

команды определения доступа к данным (DCL – Data Control Language): GRANT, REVOKE;

команды управления транзакциями (TCL – Transaction Control Language): COMMIT, ROLLBACK.

Заметим, что в базовом языке SQL отсутствуют управляющие операторы: переходы, циклы и т. п. – вследствие чего SQL не является языком программирования. Связано это с первоначальной идеологией реляционных баз данных, когда предполагалось, что при работе с базой данных конечный пользователь будет непосредственно вводить команды SQL в консольном режиме. Однако усложнение структуры баз данных привело к необходимости программной поддержки SQL. В современных СУБД она обеспечивается как с помощью процедурных расширений языка, так и посредством использования реализующих команды SQL библиотек для стандартных языков программирования.

Каждый оператор SQL состоит из имени оператора и одной или нескольких фраз, начинающихся с ключевого слова и содержащих различные языковые конструкции. Ключевое слово первой фразы является именем всего оператора. Для описания операторов будем использовать следующие обозначения:

<…> – описание какого-либо элемента оператора;

[…] – необязательная фраза или элемент;

[,…n] – необязательное повторение ноль или более раз;

{…|…|…} – обязательный выбор из списка возможностей.

Хотя ключевые слова можно записывать в любом регистре, будем для выразительности записывать их прописными буквами.

Для удобства описания операторов SQL будем считать, что эти операторы (команды) вводятся в консольном режиме: текст команды набирается на клавиатуре, а результат в виде таблицы выводится на экран. Такой режим обеспечивается в каждой системе управления базами данных с помощью специальной утилиты. Напомним, что в СУБД SQL Server она называется Management Studio. Форматы представления выходных таблиц в разных СУБД могут значительно отличаться, поэтому в примерах их описание будет приводиться в достаточно общем виде.

3.2. Демонстрационная база данных

Для иллюстрации основных возможностей языка SQL мы будем использовать базу данных «Факультет», состоящую из нескольких таблиц. Состав этой базы данных будем описывать по мере необходимости и вначале опишем только одну таблицу «Студент», содержащую персональные сведения о студентах. Структуру таблиц будем наглядно изображать в следующем виде.

Stud

nsb	fam	im	ot	grp	dr	gor	adr	stip

Названия столбцов означают следующее:
nsb – номер студенческого билета;
fam – фамилия;
im – имя;
ot – отчество;
grp – академическая группа;
dr – дата рождения;
gor – город, в котором живет студент;
adr – адрес в этом городе;
stip – размер стипендии.

Как имена таблиц, так и имена столбцов мы будем обозначать латинскими буквами, поскольку, хотя некоторые СУБД и допускают в этом качестве кириллические имена, это значительно снижает переносимость баз данных на другие платформы.

Пока что не будем детально описывать типы данных, содержащихся в столбцах таблицы, ограничившись лишь общим описанием: столбцы nsb, fam, im, ot, grp, gor и adr имеют символьный

тип, причем nsb и grp – постоянной длины, а остальные – переменной; dr имеет тип даты, stip – числовой тип.

Естественным претендентом на роль первичного ключа является столбец nsb, поскольку номера студенческих билетов у всех студентов различны. Однако, как уже отмечалось, язык SQL не требует обязательного наличия первичного ключа, при необходимости формируя ключевые поля динамически. Мы обсудим этот вопрос в разделе, посвященном индексам базы данных.

3.3 Система управления базами данных Microsoft SQL Server

Описывая операторы SQL, логически правильно было бы начать с операторов создания базы данных, создания таблиц, наполнения их содержимым и только потом переходить к операторам извлечения данных. Однако с практической точки зрения более удобно начать с наиболее важного и сложного оператора SQL – оператора SELECT.

В качестве базовой СУБД мы будем использовать SQL Server 2008 Express (русская версия), которую можно свободно скачать с сайта компании Microsoft. Выбор этой СУБД объясняется, в первую очередь, хорошей поддержкой русского языка, включающей также электронную документацию на русском языке.

3.4. Оператор Select

3.4.1. Общая структура

Оператор SELECT предназначен для извлечения информации из базы данных и позволяет делать выборку одной или нескольких строк или столбцов из одной или нескольких таблиц. При этом результатом выполнения всегда является таблица, и, даже если это только одно число, все равно оно рассматривается как таблица с одной строкой и одним столбцом.

Выполнение оператора SELECT не изменяет хранящихся в базе данных, однако в момент его выполнения запрашиваемые данные обычно блокируются от изменений.

Полное описание оператора SELECT достаточно сложно, однако его основную структуру можно выразить следующим образом:

SELECT [DISTINCT] {*|<результирующий список>}
FROM <источники данных>

[WHERE <условие отбора>]
[GROUP BY <список группировки>]
[HAVING <условие отбора>]
[ORDER BY <список сортировки> [{ASC|DESC}]]

Если результатом выполнения двух или более операторов SELECT являются таблицы одинаковой структуры, то их можно объединить в единую таблицу, разместив между этими операторами ключевое слово UNION. При этом дублирующиеся строки в результирующей таблице удаляются. Фразу ORDER BY в этом случае следует ставить после всех операторов объединения.

Аналогичным образом можно выполнить над результатами выполнения запросов теоретико-множественные операции пересечения и разности, использовав вместо ключевого слова UNION ключевые слова INTERSECT и EXCEPT.

Простейшим оператором SQL является оператор вывода полного содержимого какой-либо таблицы. В нашей демонстрационной базе данных таким будет оператор

SELECT * FROM Stud,

в котором * обозначает список всех столбцов выводимой таблицы, а Stud – имя этой таблицы.

3.4.2. Вывод отдельных столбцов

Очень часто строки таблицы достаточно велики и не помещаются в одной строке экрана. Чтобы просмотреть не все, а только некоторые столбцы таблицы, надо указать имена этих столбцов в качестве результирующих:

SELECT nsb,fam,im,ot,grp FROM Stud.

Порядок столбцов в итоговой таблице определяется порядком столбцов в операторе SELECT, а не порядком в описании исходной таблицы. Так, если мы хотим вначале выводить номер группы, то оператор SELECT можно записать следующим образом:

SELECT grp,nsb,fam,im,ot FROM Stud.

Уменьшение количества выводимых столбцов может привести к появлению в итоговой таблице дублирующихся строк. Например, если, желая вывести список городов, в которых проживают студенты, мы введем запрос

SELECT gor FROM Stud,

то получим таблицу с одним столбцом, количество строк в которой будет таким же, как в исходной таблице, а поскольку коли-

чество городов проживания студентов гораздо меньше, чем самих студентов, то большинство наименований городов будет повторяться. Чтобы исключить повторяющиеся строки, следует использовать ключевое слово DISTINCT:

SELECT DISTINCT gor FROM Stud.

Выполнение запроса с исключением повторяющихся строк происходит медленнее, чем без ключевого слова DISTINCT, поэтому если нет оснований полагать, что в итоговой таблице будут повторяющиеся строки (а чаще всего так и бывает), то лучше это ключевое слово не использовать.

3.4.3. Вычисляемые столбцы

В некоторых случаях является желательным выводить в итоговой таблице не только столбцы, содержащие информацию, непосредственно хранящуюся в базе данных, но и столбцы, содержащие значения, вычисляемые по значениям в других столбцах, либо содержащие постоянную информацию. Примерами могут являться столбцы, содержащие стоимость товара, вычисляемую как результат произведения количества товара на стоимость единицы товара, или столбцы, содержащие во всех строках текущую дату или слова «Итого», «Всего», «Сумма» и т. д.

SQL предоставляет возможность формирования таких столбцов, для чего достаточно просто записать в операторе SELECT вместо имени столбца соответствующее выражение, например:

SELECT quantity, cost, quantity*cost FROM current .

Допускаются и более сложные варианты; так, например, в системе SQL Server оператор

SELECT 'Время: '+CONVERT(char(30),CURRENT_TIMESTAMP)

выдаст преобразованные к строке длиной 30 символов текущие дату и время, дополненные префиксом «Время:».

В этом примере хотя результат представляет собой один-единственный элемент строкового типа, он все равно рассматривается как таблица из одного столбца и одной строки.

В общем случае для построения арифметического выражения можно обычным образом использовать числовые константы, имена столбцов, знаки арифметических действий и круглые скобки. Сложнее обстоит дело со строковыми выражениями. В минимальном варианте стандарт SQL допускает лишь использование простейших констант типа 'Всего' или '10.12.1947', однако в

каждой системе управления базами данных имеются расширения, позволяющие конструировать достаточно сложные выражения строкового типа. Пример для SQL Server приведен выше.

Особенностью вычисляемых столбцов является то, что имя заголовка такого столбца либо пусто, либо присваивается системой. В то же время в ряде случаев необходимо присвоить столбцу то или иное конкретное имя. Для этого достаточно соответствующее выражение дополнить фразой AS <имя> :

SELECT quantity, cost, quantity*cost AS total FROM current.

Фразу AS можно применять не только для вычисляемых столбцов, но и во всех других случаях, когда необходимо изменить имя столбца. Чаще всего это требуется сделать в случае применения операций объединения, пересечения или разности таблиц, поскольку в этих случаях структуры таблиц (включая имена столбцов) должны быть одинаковы.

3.4.4. Выборка отдельных строк

Как уже отмечалось, реляционные базы данных наиболее приспособлены для обработки больших массивов однородной информации. Поэтому таблицы, как правило, содержат большое количество строк, а оператор SELECT выводит лишь часть строк таблицы. Для этого используется фраза

WHERE <условие>,

в которой в качестве условия записывается некоторое логическое выражение, и строка выводится на экран, только если соответствующее этой строке значение выражения истинно. Примером может служить оператор

```
SELECT grp,nsb,fam,im,ot FROM Stud
WHERE grp = 'MO-33'.
```

В результате выполнения этого оператора на экран будут выведены сведения только о студентах группы МО-33.

Способы построения выражений уже упоминались при обсуждении вычисляемых столбцов, однако при построении логических выражений используются дополнительные средства. По-прежнему основными «кирпичиками», из которых строятся выражения, являются имена столбцов и константы. Числа записываются так же, как и в большинстве традиционных языков программирования, а символьные строки и даты заключаются в одиночные кавычки, причем

дата записывается в национальном формате (в том, на который настроена система). Примеры констант:

'это символьная строка', -385, '25.10.2009'.

Выражения строятся из имен столбцов и констант с применением знаков арифметических операций +, –, *, /, круглых скобок, операций отношения =, <, >, <=, >=, <>, логических операций NOT, AND, OR, а также некоторых специфических для SQL конструкций. Отметим, что старшинство операций не всегда очевидно, поэтому в сомнительных случаях лучше не жалеть круглых скобок.

Из специфических конструкций опишем лишь наиболее часто применяемые фразы IN, LIKE и трехзначную логику.

Фраза IN применяется, когда нужно отобрать строки, у которых значение в каком-либо столбце принадлежит заданному списку, например

```
SELECT grp,nsb,fam,im,ot FROM Stud
 WHERE grp IN ('МО-33','ПИЭ-21','ИВТ-52').
```

Фраза like (а также not like) наиболее часто используется при сравнении строк, не совпадающих по длине. Соответствующее условие записывается в виде

WHERE <строка> LIKE <шаблон>.

Под шаблоном здесь понимается строка, содержащая символы подстановки % и _, причем % обозначает любую последовательность символов, а знак подчеркивания _ – в точности один символ (в файловой системе Windows аналогичную роль играют символы * и ?). Например, шаблону 'Иван%' соответствуют строки 'Иванов', 'Иванищев', 'Иванченко' и т. п.

В случае, когда в шаблон требуется включить сами знаки % или _, следует дополнить условие фразой ESCAPE '<экранирующий символ>' и поставить экранирующий символ перед знаком подстановки. Например, условие

```
WHERE reslt LIKE '100\%' ESCAPE '\'
```

отберет только строки, в которых значение reslt равно 100%.

Важной особенностью языка SQL является использование т. н. трехзначной логики, под которой в этом контексте понимается возможность присвоения некоторым ячейкам таблиц пустых значений (null), которые в условных выражениях обрабатываются по особым правилам. При этом пустое значение может находиться (если при описании таблицы не оговорено противное) в любой ячейке таблицы, независимо от ее типа.

Основной принцип использования пустых значений состоит в том, что если выражение содержит элемент с пустым значением, то всему выражению также присваивается пустое значение. Соответственно, результат вычисления условного выражения может принимать одно из трех значений: истина, ложь или пусто, а строка включается в результирующий набор, только если это значение истинно.

В некоторых случаях целесообразно сразу выяснить, является ли то или иное значение пустым. Этой цели служат операции сравнения IS null, IS NOT null, как, например, в условном выражении

WHERE adr IS NOT null.

Заметим, что это не то же самое, что adr <> null, поскольку в соответствии с вышеописанными правилами последнее выражение имеет пустое значение, и, соответственно, в результирующий набор не войдет ни одной строки, в то время, как в первом случае результирующий набор будет содержать строки с непустым значение adr.

В языке Transact-SQL (СУБД SQL Server) весьма полезной для работы с пустыми значениями является функция ISnull(<выражение1>, <выражение2>), которая проверяет, является ли <выражение1> пустым, и если это так, то возвращает <выражение2>, в противном же случае – <выражение1>. Аналогичные функции есть и в других СУБД.

3.4.5. Сортировка результатов запроса

Результат выполнения запроса может быть (что чаще всего и делается) отсортирован по одному или нескольким столбцам. Для этого применяется фраза ORDER BY:

ORDER BY <список столбцов> .

Результат сортируется по значениям первого столбца, при совпадающих значениях первого – по значениям второго и т. д. По умолчанию сортировка производится в порядке возрастания значений, однако если после имени столбца поставить уточнитель DESC, то для этого столбца сортировка будет вестись в порядке убывания. Примером может служить запрос

SELECT * FROM stud ORDER BY stip DESC, fam.

Сортировка будет вестись в порядке убывания размеров стипендии, а в случае одинаковой стипендии – в алфавитном порядке фамилий.

Если для получения результирующей таблицы используется операция объединения UNION, то фразу ORDER BY следует ставить после последнего оператора SELECT, входящего в объединение.

3.4.6. Статистические функции

Мы уже видели, что SQL позволяет выполнять арифметические операции над ячейками одной строки при формировании вычисляемых столбцов. Можно также выполнять и действия над ячейками одного столбца, однако список допустимых действий достаточно ограничен. Это вычисление минимального, максимального и среднего арифметического значений в столбце, суммы значений по столбцу, а также количества элементов в столбце. Соответствующие возможности обеспечиваются т. н. статистическими (называемыми также итоговыми или агрегатными) функциями MIN, MAX, AVG, SUM и COUNT.

Например, запрос

SELECT SUM(stip) FROM stud

выдаст в качестве результата общую сумму назначенной стипендии.

Особенностью использования арифметических (первых четырех) функций является то, что при их расчете строки, содержащие пустое значение (null) в соответствующем столбце, не принимаются во внимание. Это особенно следует иметь в виду при расчете среднего значения, поскольку оно вычисляется только по строкам с непустыми значениями.

Функция COUNT имеет расширенный синтаксис и может быть использована в трех вариантах: COUNT (<столбец>), COUNT (DISTINCT <столбец>) и COUNT(*). В первом варианте вычисляется, аналогично другим функциям, количество строк с непустым значением в соответствующем столбце, во втором – количество различных непустых значений в этом столбце, а в третьем – общее количество строк в таблице (включающих и пустые значения).

3.4.7. Работа с группами

В чистом виде итоговые функции применяются довольно редко. Гораздо чаще они используются для расчета промежуточных итогов, получаемых применением статистических функций к отдельным группам строк таблицы. Группировка строк производится по принципу равенства значений для этих строк в одном

или нескольких столбцах, называемых столбцами группировки. Типичным примером является запрос

```
SELECT stip, COUNT(stip), SUM(stip) FROM stud
 GROUP BY stip .
```

Результирующая таблица содержит возможные размеры стипендий, количество студентов, получающих такую стипендию, и общую сумму, выделенную на стипендии такого размера.

В запросе с группировкой перечень выводимых столбцов должен содержать все столбцы группировки (и никакие другие), а также одну или несколько статистических функций, не обязательно относящихся к столбцам группировки. Например, запрос

```
SELECT grp, SUM(stip) FROM stud GROUP BY grp
```

выведет размер стипендиального фонда для каждой студенческой группы.

Отметим одну особенность обработки пустых значений при группировке. Если столбец группировки содержит пустые значения, то все строки, имеющие пустые значения в этом столбце, помещаются в одну группу.

Часто бывает нужным вывести не все промежуточные итоги, а только какую-либо их часть. Для этого применяется фраза HAVING <условие>, где условие – это логическое выражение, составленное из статистических функций и/или столбцов группировки. Вот примеры таких запросов:

```
SELECT grp, SUM(stip) FROM stud GROUP BY grp
 HAVING AVG(stip)>1500 ;
SELECT grp, SUM(stip) FROM stud GROUP BY grp
 HAVING grp LIKE 'МО%' .
```

Допускается использование с одном запросе как фразы HAVING, так и фразы WHERE. Так запрос

```
SELECT grp, COUNT(stip) FROM stud
 WHERE gor='Ярославль' AND stip>0
 GROUP BY grp
 HAVING grp LIKE 'МО%'
```

выведет количество ярославских студентов, получающих стипендию, в группах МО.

В заключение приведем нетривиальный пример использования операции объединения: получение результирующей таблицы с промежуточными итогами:

```
SELECT grp, ' '+fam AS fam, stip FROM stud
```

```
UNION
SELECT grp, 'Всего:' AS fam, SUM(stip) AS stip
 FROM stud
 GROUP BY grp
ORDER BY grp, fam .
```

Этот пример не вполне соответствует стандарту SQL, и, хотя он работает в СУБД SQL Server, в других системах это может оказаться не так.

3.4.8. Многотабличные запросы

До сих пор мы рассматривали запросы только с одной исходной таблицей. Однако основную роль при работе с большими базами данных играют запросы с несколькими исходными таблицами.

Для иллюстрации многотабличных запросов дополним нашу демонстрационную базу данных еще одной таблицей «Сессия», содержащей сведения о результатах экзаменов. Ее структура имеет следующий вид.

Sess

nsb	*disc*	*prep*	*ball*

Названия столбцов означают следующее:

nsb – номер студенческого билета;

disc – дисциплина;

prep – фамилия преподавателя;

ball – оценка.

Синтаксис многотабличного запроса почти не отличается от однотабличного: во фразе FROM указывается не одна, а несколько таблиц-источников. Однако, поскольку в разных таблицах могут быть столбцы с одинаковыми именами (в нашем случае – *nsb*), имена столбцов в запросах следует писать с именем таблицы в виде префикса: *stud.nsb, sess.nsb*. В частности, для обозначения всех столбцов таблицы можно использовать звездочку: *stud.**. Если же имя столбца уникально в используемых таблицах, то префикс можно не указывать.

На первый взгляд простейший запрос с двумя таблицами мог бы иметь вид

```
SELECT * FROM stud, sess .
```

Такой запрос, тем не менее, практически бесполезен, поскольку выдает произведение таблиц, части которого совершенно не связаны друг с другом. В общем случае такая связь может быть установлена включением в запрос фразы WHERE <условие>, где условие – логическое выражение, содержащее значения из разных таблиц. Чаще всего такое выражение содержит условие равенства значений совпадающих по смыслу (но не обязательно по имени) столбцов. В качестве примера приведем запрос

```
SELECT stud.*, disc, prep, ball FROM stud, sess
WHERE stud.nsb = sess.nsb .
```

Этот запрос выводит полные результаты сессии, причем для каждой дисциплины, сдававшейся студентом, полностью приводится персональная информация об этом студенте.

Наряду с условием связи таблиц во фразе WHERE могут присутствовать и другие условия отбора строк результирующей таблицы. Например, запрос

```
SELECT grp, fam, disc, ball FROM stud, sess
WHERE (stud.nsb = sess.nsb) AND (grp='МО-33')
```

выводит информацию о результатах сдачи сессии студентами группы МО-33.

С точки зрения «правильного» проектирования запросов не вполне корректно смешивать в одном запросе условия связи таблиц и условия, описывающие логику запроса. В дальнейшем (при обсуждении соединений – join) мы увидим, как эта проблема решена в стандарте SQL92.

Мы рассмотрели простейший случай двух таблиц с одним столбцом связи. Понятно, что увеличение количества таблиц и столбцов связи не вносит принципиальных изменений в методику построения многотабличных запросов.

В некоторых случаях при построении многотабличных запросов оказывается полезным использование псевдонимов (алиасов) таблиц. Псевдоним представляет собой идентификатор, который через пробел записывается после имени таблицы во фразе FROM, после чего он используется в качестве префикса имен столбцов. Приведем пример, когда использование псевдонимов позволяет решить задачу, которая без их использования решалась бы гораздо сложнее.

Пусть у нас задана таблица сотрудников предприятия:

Pers

tabn	*fio*	*tabnr*

Названия столбцов означают следующее:

tabn – табельный номер сотрудника;

fio – фамилия и инициалы сотрудника;

tabnr – табельный номер руководителя сотрудника.

Требуется вывести таблицу из двух столбцов, которые содержат соответственно фамилию и инициалы сотрудников и фамилию и инициалы руководителей. Для этого можно сформировать запрос

SELECT a.fio AS fio, b.fio AS fior FROM pers a, pers b

WHERE a.tabnr = b.tabn .

Его особенностью является то, что таблица pers перекрестно соединяется со своей копией, что и решает поставленную задачу.

3.4.9. Внешние соединения

Мы уже отмечали, что не очень логично смешивать в одной фразе WHERE условия связи таблиц и условия, описывающие логику обработки данных. Кроме того, описанный в предыдущем пункте способ связи таблиц обладает тем недостатком, что в результирующую таблицу попадают данные только из тех строк исходных таблиц, для которых есть соответствующие строки в другой таблице. Так, если студент в приводившемся примере пока еще не сдавал ни одного экзамена, то в итоговой таблице информация о нем вообще будет отсутствовать, хотя логичнее было бы информацию о студенте разместить в этой таблице, заполнив информацию об экзаменах пустыми значениями.

Обе этих проблемы решаются способом, предложенным в стандарте SQL92 и называемым соединением таблиц (JOIN). Смысл его заключается в создании на основе двух таблиц нового объекта, называемого соединением двух таблиц и который может размещаться на месте исходной таблицы во фразе FROM. Синтаксис соединения выглядит следующим образом:

<левая таблица> <тип соединения> JOIN <правая таблица>

ON <условие соединения> .

В качестве типа соединения могут быть использованы LEFT OUTER (левое внешнее), RIGHT OUTER (правое внешнее), FULL OUTER (полное внешнее), INNER (внутреннее). При этом слова INNER и OUTER могут быть опущены. Рассмотренный ранее ва-

риант соединения, при котором из исходных выбираются только соответствующие друг другу строки, называется внутренним соединением, а приведенный пример может быть записан следующим образом:

```
SELECT grp, fam, disc, ball
 FROM stud INNER JOIN sess ON stud.nsb = sess.nsb
 WHERE grp='MO-33' .
```

При этом в результирующей таблице будут присутствовать только те студенты, которые сдали хотя бы один экзамен. Если же мы хотим включить в итоговую таблицу всех студентов, запрос должен быть записан как

```
SELECT grp, fam, disc, ball
FROM stud LEFT OUTER JOIN sess ON stud.nsb = sess.nsb
WHERE grp='MO-33' .
```

В итоговой таблице будут находиться все значения grp и fam левой исходной таблицы, причем при отсутствии соответствующих строк правой таблицы значения disc и ball будут пустыми.

Для иллюстрации возможных вариантов соединений рассмотрим простой пример. Пусть имеются таблицы

t1

a1	*a2*
1	a
2	s
4	d

t2

a1	*a3*
2	f
3	g
4	h

Тогда запрос

```
SELECT t1.a1, t2.a1, a2, a3 FROM t1 <тип соединения>
JOIN t2 ON t1.a1 = t2.a1,
```

где тип соединения – INNER (внутреннее), LEFT OUTER (левое внешнее), RIGHT OUTER (правое внешнее) или FULL OUTER (полное внешнее), выдаст следующие результаты:

Внутреннее				Левое				Правое				Полное			
t1.a1	t2.a1	a2	a3	t1.a1	t2.a1	a2	a3	t1.a1	t2.a1	a2	a3	t1.a1	t2.a1	a2	a3
2	2	s	f	1	null	a	null	2	2	s	f	1	null	a	null
4	4	d	h	2	2	s	f	null	3	null	g	2	2	s	f
				4	4	d	h	4	4	d	h	4	4	d	h
												null	3	null	g

3.4.10. Вложенные запросы

Часто приходится строить запросы, для которых необходима информация, получаемая с помощью других запросов. Простейшим примером является запрос на вывод из таблицы Stud

списка студентов, получающих максимальную стипендию. Очевидно, что для этого необходимо выполнить сначала запрос

SELECT MAX(stip) FROM stud

и запомнить или записать полученное значение, а затем запрос

SELECT grp, fam, im, ot FROM Stud

WHERE stip = <максимальная стипендия>

для получения требуемого списка студентов. Мы уже отмечали, что SQL не является языком программирования и в стандартном варианте в нем отсутствуют переменные для запоминания промежуточных значений. Заменой этому является возможность записывать в выражениях, используемых во фразах WHERE и HAVING, операторы SELECT, результатом выполнения которых является единственное значение, точнее, таблица из одной строки и одного столбца. Такие операторы можно использовать везде, где в выражениях может быть записано имя столбца или константа. Предыдущий пример может быть записан в виде

SELECT grp, fam, im, ot FROM Stud

WHERE stip = (SELECT MAX(stip) FROM Stud) .

Возможности применения вложенных запросов (подзапросов) в SQL немного шире, чем использование только одиночных значений. Сохраняя основную идею записи подзапросов только в WHERE или HAVING, они допускают небольшие отступления от принципа «один столбец и одна строка».

Простейшим вариантом является принадлежность значения выражения множеству значений результата выполнения запроса, например, запрос

SELECT grp, fam, nsb, stip FROM Stud

WHERE grp IN (SELECT grp FROM Stud GROUP BY grp

HAVING AVG(stip) > (SELECT AVG(stip) FROM Stud))

выдаст список студентов тех групп, средняя стипендия в которых превышает общую среднюю стипендию.

Приведенный запрос демонстрирует также возможность многократного вложения запросов – в нашем случае количество «матрешек» равно трем.

Тот же результат, что и в предыдущем случае, может быть достигнут и с помощью оператора

SELECT grp, fam, nsb, stip FROM Stud

WHERE grp = ANY (SELECT grp FROM Stud GROUP BY grp

HAVING AVG(stip) > (SELECT AVG(stip) FROM Stud)),

условие в котором означает, что значение grp должно совпадать с каким-нибудь из результирующих значений внутреннего запроса. Вместо ANY можно использовать ключевое слово SOME.

Ключевое слово ANY чаще применяется при сравнении выражения и подзапроса на неравенство. Например, запрос

```
SELECT grp, fam, nsb, stip FROM Stud
WHERE stip > ANY (SELECT DISTINCT stip FROM Stud)
```

выдаст список студентов, получающих стипендию больше какой-либо из возможных (т. е. в нашем случае – больше минимальной). При этом предполагается, что неназначение стипендии обозначается пустым значением.

Если требуется, чтобы значение выражения было больше любого значения, выдаваемого подзапросом, то вместо ANY следует использовать ключевое слово ALL. Так, запрос

```
SELECT grp, fam, nsb, stip FROM Stud
WHERE stip >
ALL (SELECT DISTINCT AVG(stip) FROM Stud GROUP BY grp)
```

выдаст список студентов, получающих стипендию больше средней в любой группе.

Еще одной возможностью, используемой в подзапросах, является проверка на существование или несуществование результирующих строк подзапроса. Следующий запрос выводит список студентов, которые имеют хотя бы одну неудовлетворительную оценку:

```
SELECT grp, fam, im, ot FROM Stud
 WHERE EXISTS (SELECT * FROM Sess
 WHERE (Sess.nsb=Stud.nsb) AND ball=2) .
```

До сих пор мы применяли вложенные запросы во фразах WHERE и HAVING. Однако, как мы уже видели, выражения могут использоваться при создании вычисляемых столбцов, и в этом случае также могут использоваться вложенные запросы. Наиболее типичное их применение демонстрируется запросом

```
SELECT grp, fam, im, ot,
 (SELECT COUNT(*) FROM Sess WHERE (Sess.nsb=Stud.nsb))
 FROM Stud ,
```

который для каждого студента выводит количество сданных им экзаменов.

3.5. Внесение изменений в базу данных

Мы рассмотрели основные возможности оператора SELECT, который производит выборку (чтение) данных из базы данных и не изменяет ее содержимое. Перейдем теперь к операторам, осуществляющим изменение данных, хранящихся в базе данных (часто говорят «производят запись», что не совсем точно). Применительно к реляционной модели данных это операторы INSERT INTO, добавляющие строки в таблицы базы данных, DELETE, удаляющие строки, и UPDATE, вносящие частичные или полные изменения в строки таблиц базы данных.

3.5.1. Вставка строк

Оператор вставки строк INSERT INTO имеет две разновидности: для вставки одиночной строки и для вставки группы строк.

Оператор вставки в таблицу одной строки записывается как

INSERT INTO <таблица> [(<список столбцов>)]

VALUES (<список значений>) .

Список значений должен по количеству элементов и по типам значений соответствовать списку столбцов. Если список столбцов не приведен, то это означает, что вставка ведется во все столбцы таблицы в том порядке, в котором они описаны в описании таблицы. Крайне не рекомендуется опускать этот список, поскольку любое изменение структуры таблицы потребует изменения и в операторах INSERT INTO, что противоречит требованиям логической независимости данных.

Если в списке столбцов некоторые столбцы таблицы не приведены, то при отсутствии дополнительных указаний (значений по умолчанию, которые будут рассмотрены при описании оператора CREATE TABLE) в соответствующие позиции вставляемой строки будут записаны пустые значения (null). С другой стороны, при наличии таких указаний можно явно присвоить той или иной позиции вставляемой строки пустое значение, записав его в списке значений, например:

```
INSERT INTO Stud (nsb,fam,im,ot,grp,gor) VALUES
 (123456,'Иванов','Петр','Сергеевич','ПИЭ-33',null) .
```

Здесь полю gor значение null присваивается явным образом, а полям dr, adr, stip – неявно, поскольку они отсутствуют в списке столбцов.

Для множественной вставки строк в таблицу используется оператор

INSERT INTO <таблица> [(<список столбцов>)]
<оператор SELECT> ,

где указанный оператор SELECT обладает тем свойством, что столбцы его результирующей таблицы по количеству элементов и по типам значений соответствуют списку столбцов таблицы, в которую производится вставка. Имена столбцов результирующей таблицы оператора SELECT при этом несущественны. В частности, они могут быть вычисляемыми, содержащими агрегатные функции и т. д.

При использовании оператора SELECT в операторах модификации базы данных возникает один тонкий момент, связанный с тем, что оператор SELECT может содержать условие во фразе WHERE или HAVING, которое в случае присутствия в этом условии модифицируемой таблицы будет иметь различный смысл до вставки строк, в процессе вставки и по ее окончании. Чтобы в этом случае избежать двусмысленности, принимается, что все условия вычисляются до начала вставки строк и уже с учетом этого обстоятельства выполняется оператор SELECT.

Отметим, что, помимо указанных двух способов вставки строк в таблицу, в каждой конкретной СУБД существуют и другие способы. В частности, практически в каждой СУБД существуют утилиты массовой загрузки (миграции) данных, которые заполняют таблицы базы данных информацией из заранее подготовленных файлов в текстовых форматах, электронных таблиц, а также из других баз данных. Однако такая загрузка происходит достаточно редко, чаще всего при запуске в эксплуатацию новой информационной системы.

3.5.2. Удаление строк

Оператор удаления строк является простейшим из операторов работы со строками. Он имеет следующий формат:

DELETE FROM <таблица> [WHERE <условие отбора>]

Если условие не указано, то этот оператор удаляет из таблицы все строки, т. е. производит очистку таблицы, сохраняя ее структуру. Если же необходимо удалить из таблицы только часть строк, то используется условие отбора, которое представляет собой логическое выражение, составляемое по тем же правилам, что и для оператора SELECT. В частности, оно может содержать вложенные операторы SELECT. Для оператора DELETE применяется тот же принцип, что и в предыдущем пункте: все условия вычисляются до начала выполнения оператора.

3.5.3. Замена строк

Оператор замены данных в строках таблицы более сложен и имеет следующий вид:

UPDATE <таблица> SET <столбец1> = <значение1> [,…]
[WHERE <условие отбора>]

Оператор заменяет данные в указанных столбцах приводимыми значениями. Условие определяет строки, в которых будет производиться такая замена (при его отсутствии замена производится во всех строках таблицы). Как и в предыдущих случаях, условие отбора может содержать вложенные операторы SELECT. При этом в условии могут присутствовать и другие таблицы.

3.6. Создание таблиц

До сих пор мы рассматривали действия с данными в таблицах, предполагая, что таблицы уже существуют в базе данных, так же, как и сама база данных. Перейдем теперь к рассмотрению операторов языка SQL, предназначенных для действий с таблицами в целом, а также для работы с другими объектами, содержащимися в базе данных.

Почти для всех таких объектов (включая таблицы) используются единые обозначения операторов: CREATE – создание объекта, DROP – удаление объекта, ALTER – изменение свойств. После имени оператора следует тип объекта (например, TABLE – таблица), затем имя объекта и необходимые параметры.

Попутно затронем вопрос о создании объекта верхнего уровня – базы данных. Хотя в некоторых СУБД и существует оператор CREATE DATABASE, чаще база данных создается в графическом, а не текстовом режиме. Связано это с тем, что создание базы данных – операция достаточно редкая и исполняется посредством консольного, а не программного доступа. Для консоли же графический режим является более предпочтительным. Кроме того, форматы баз данных различаются настолько сильно, что стандартизовать оператор CREATE DATABASE не представляется возможным. Поэтому в стандартах языка он отсутствует, а роль средства, объединяющего таблицы в единое целое, играет оператор CREATE SCHEMA. База данных играет в этом случае роль физического хранилища объектов, а их логическая связь обеспечивается схемами информационных систем.

Если база данных уже создана, то таблицы в ней создаются с помощью оператора CREATE TABLE, формат которого достаточно сложен. В простейшем варианте он выглядит так:

CREATE TABLE <имя таблицы> (<имя столбца> <тип столбца> [,…])

Используемые типы столбцов очень сильно зависят от применяемой СУБД, однако некоторые из них общеприняты. Это int или integer – целый тип, char(n) – символьная строка постоянной длины n, varchar(n) – символьная строка переменной длины до n символов, date – дата. При использовании других типов следует изучить соответствующие разделы документации по применяемой СУБД. Подобная ситуация весьма затрудняет адекватный перенос базы данных, работающей под одной СУБД, в систему с другой СУБД. Чаще всего для этого необходимо использовать или разрабатывать специальные программные средства.

В качестве примера приведем перечень типов данных, используемых в СУБД SQL Server 2008 Express.

Числовые	Символьные	Дата/Время	Специальные
tinyint	char(n)	date	geography
smallint	varchar(n)	datetime	geometry
int	text	datetime2(n)	hierarchyid
bigint	nchar(n)	datetimeoffset(n)	sql_variant
decimal(n,m)	nvarchar	smalldatetime	timestamp
numeric(n,m)	ntext	time(n)	uniqueidentifier
real	binary(n)		xml
float	varbinary(n)		
bit	image		
smallmoney			
money			

Аналогичная таблица для СУБД MySQL 5.5 выглядит совершенно иначе.

Числовые	Символьные	Дата/Время	Специальные
tinyint	char	date	geometry
smallint	varchar	datetime	point
mediumint	tinytext	timestamp	linestring
nt	text	time	polygon
bigint	mediumtext	year	multipoint
decimal	longtext		multilinestring
float	binary		multipolygon
double	varbinary		geometrycollection
real	tinyblob		

bit	mediumblob		
bool	blob		
serial	longblob		
	enum		
	set		

Нетрудно видеть, что общие наименования типов составляют очень небольшое множество. Более того, тип timestamp вообще имеет разный смысл и находится в разных группах. Наоборот, близкие по смыслу типы имеют совершенно непохожие наименования, например image в SQL Server и blob в MySQL. Поэтому при практической разработке структуры базы данных просто необходимо ознакомиться с системой типов данных используемой СУБД.

Перейдем теперь к более сложным элементам оператора CREATE TABLE. Они делятся на две группы: уточненные спецификации столбцов и определения ограничений – синтаксических конструкций, которые позволяют запретить те или иные значения или комбинации значений в столбцах или в таблицах в целом.

Расширенный формат оператора CREATE TABLE может быть описан в следующем виде:

CREATE TABLE <имя таблицы> (<определение столбца>
[, {<определение столбца>|<табличное ограничение>}…)

Обычно вначале записываются все определения столбцов, а затем все табличные ограничения.

Определение столбца задается в виде

<имя столбца> <тип данных> [DEFAULT <константа>]
[<ограничение столбца>[…]]

Константа в этом случае задает значение в соответствующем столбце, если при вставке строки этот столбец не указан в списке аргументов команды INSERT INTO.

К ограничениям столбца относятся следующие:

NOT null – запрет пустых значений

PRIMARY KEY – первичный ключ;

UNIQUE – альтернативный ключ;

CHECK (<условие>) – условие проверки;

ограничения связи таблиц.

Ограничения PRIMARY KEY и UNIQUE запрещают операции вставки и замены строк, при которых в соответствующем столбце могут появиться дублирующиеся значения. При попытке выполнения таких действий СУБД инициирует сообщение об

ошибке. Разница между двумя этими типами ограничений заключается в том, что первичный ключ используется также при описании связи двух таблиц (см. далее).

Условие проверки запрещает выполнение операций вставки или замены, при которых это условие окажется ложным.

Наиболее сложным из ограничений столбца является ограничение связи таблиц, иногда называемое ограничением целостности. Для его описания столбец в создаваемой командой CREATE TABLE таблице, которая в этом случае выступает в качестве дочерней, сопоставляется первичному ключу родительской таблицы. Столбец же в дочерней таблице называется внешним ключом (FOREIGN KEY). Синтаксис этого ограничения следующий:

FOREIGN KEY REFERENCES <родительская таблица>
[ON DELETE {NO ACTION|CASCADE|SET null|SET DEFAULT}]
[ON UPDATE {NO ACTION|CASCADE|SET null|SET DEFAULT}]

Смысл связи таблиц состоит в том, что СУБД запрещает вставку в дочернюю таблицу строки с предлагаемым значением внешнего ключа, если в родительской таблице отсутствует строка с таким же значением первичного ключа. Так, в таблицу Sess будет запрещена вставка строки с номером студенческого билета, которого нет в таблице Stud, поскольку это означало бы сдачу экзамена несуществующим студентом.

Несколько сложнее обстоит дело с удалением и заменой строк в родительской таблице. При этих действиях некоторые строки в дочерней таблице могут оказаться «сиротами» – не иметь соответствующих им строк в родительской таблице. Возможные пути разрешения этой ситуации описываются двумя последними строками синтаксиса ограничения связи. При выборе [NO ACTION] никаких действий выполняться не будет, и «сироты» таковыми и останутся. Выбор [CASCADE] (каскадное удаление или изменение) предписывает удалить все дочерние строки одновременно с родительскими либо заменить значение в соответствующих столбцах на значение в родительской строке. [SET DEFAULT] назначает всем «сиротам» заранее определенного родителя, а [SET null] оставляет его неопределенным.

Табличные ограничения используются в тех случаях, когда ключи состоят из нескольких столбцов. В этом случае список столбцов указывается в круглых скобках после названия типа

ключа. Табличное ограничение CHECK может включать несколько столбцов.

Важной особенностью ограничений является то, что им может быть присвоено имя с помощью конструкции CONSTRAINT <имя ограничения>, используемой перед описанием ограничения. Если же эта конструкция не использована, то СУБД присваивает ограничению некоторое собственное системное имя, которое в принципе доступно для просмотра. В дальнейшем имя ограничения может быть использовано при изменении описания таблицы командой ALTER TABLE.

В качестве примера приведем описание использовавшихся нами ранее таблиц Stud и Sess.

```
CREATE TABLE Stud (
 nsb int PRIMARY KEY,
 fam varchar(20) NOT null,
 im varchar(20) DEFAULT '',
 ot varchar(20) DEFAULT '',
 grp char(10) NOT null,
 dr date CHECK (dr BETWEEN '01.01.1930'
  AND '01.01.2000'),
 gor varchar(20),
 adr varchar(100),
 stip int
)
CREATE TABLE Sess (
 nsb int ,
 disc varchar(60),
 prep varchar(20),
 ball int,
 PRIMARY KEY (nsb,disc),
 CONSTRAINT sess_stud FOREIGN KEY (nsb) REFERENCES Stud
)
```

Отметим, что, несмотря на общее единство подходов к структуре команды CREATE TABLE, конкретные ее реализации в различных СУБД могут заметно отличаться друг от друга.

3.7. Удаление таблиц и изменение их свойств

Удаление таблицы осуществляется оператором:

DROP TABLE <таблица>

Оператором DROP TABLE нельзя удалить таблицу, на которую ссылается ограничение FOREIGN KEY. Сначала следует

удалить это ограничение или дочернюю таблицу, а уже затем – родительскую таблицу.

Отметим, что при удалении всех строк в таблице с помощью оператора DELETE FROM <таблица> таблица не удаляется, а только очищается и в нее можно добавлять новые строки, не создавая ее заново.

Созданное командой CREATE TABLE описание таблицы может быть впоследствии изменено, для чего используется команда ALTER TABLE вида

ALTER TABLE <таблица> <действие>

Сразу заметим, что возможности этой команды по изменению структуры таблицы достаточно ограничены. Например, с ее помощью нельзя изменить тип данных столбца. Допускаются следующие действия:

добавление столбца: ADD <определение столбца>

удаление столбца: DROP <столбец>

Изменение определения столбца:

ALTER <столбец> {SET DEFAULT <значение>|DROP DEFAULT}

Добавление ограничения:

ADD [CONSTRAINT <имя>] <определение ограничения>

Удаление ограничения:

DROP CONSTRAINT <имя ограничения>

Если ограничению не было присвоено имя командой CREATE TABLE, то для удаления ограничения надо определить его имя с помощью соответствующей системной утилиты (своей для каждой СУБД).

3.8. Представления

Пожалуй, вторыми по значимости после таблиц объектами базы данных являются представления, называемые также виртуальными таблицами.

Представление можно рассматривать как таблицу, которая получается в результате выполнения какого-либо оператора SELECT и которой присвоено имя. В этом качестве представление может быть использовано в качестве источника данных в других операторах SELECT.

Заметим, что в персональной СУБД Access представления называются запросами на выборку.

Синтаксис оператора создания представления выглядит следующим образом:

CREATE VIEW <имя> [<список столбцов>] AS
<оператор SELECT>

Здесь SELECT – более или менее произвольный оператор этого типа. Однако в конкретных СУБД на этот оператор накладываются различные ограничения, связанные обычно с реализацией оператора SELECT в этих системах. Например, в SQL Server не разрешено использование в операторе SELECT, порождающем представление, фразы ORDER BY.

Отличие виртуальной таблицы от реальной состоит в том, что данные в том формате, который она описывает, появляются только в момент обращения к ней, когда и выполняется порождающий ее оператор SELECT. Постоянно же в базе данных хранится только текст, описывающий эту виртуальную таблицу (представление). Тем не менее представления обладают многими свойствами реальных таблиц: они могут быть проиндексированы, на них могут назначаться права использования и т. д.

В некоторых случаях, особенно при работе в многопользовательской среде, могут оказаться полезными т. н. обновляемые представления. Обновляемыми называются представления, которые могут использоваться не только в операторе SELECT, но также и операторах INSERT INTO, DELETE и UPDATE. Естественно, что при этом накладываются дополнительные условия как на оператор SELECT, порождающий представление, так и на операторы модификации данных. Простейшим из таких условий является требование, чтобы базовая таблица представления (таблица, входящая в состав источников данных порождающего оператора SELECT) была единственной, а результирующая таблица представляла собой подмножество строк и столбцов базовой таблицы, т. е. в операторе SELECT отсутствовали бы вычисляемые столбцы, статистические функции, группировки и т. п. Фактически это означает, что с помощью представления пользователь видит только часть таблицы и не подозревает о существовании других ее частей. Понятно, что при добавлении строки в такую урезанную таблицу в «невидимые» столбцы будут записываться либо значения по умолчанию, либо значения null (если это разрешено).

На самом деле в конкретных СУБД приведенное ограничение ослабляется, но описанный случай является наиболее употребительным.

3.9. Индексы

Таблицы и представления являются наиболее используемыми из объектов, хранящихся в базе данных. Однако, помимо них, база данных хранит большое количество других объектов. Достаточно сказать, что в SQL Server оператор CREATE насчитывает около полусотни разновидностей. Система MySQL гораздо скромнее, имея лишь около десятка разновидностей этого оператора. К сожалению, перечень объектов, используемых в различных СУБД, не поддается разумной классификации. Из всего их разнообразия мы рассмотрим лишь индексы, хранимые процедуры и триггеры.

Основным назначением индексов является ускорение внутренних операций, связанных с выборкой строк. Не вдаваясь в подробности, можно сказать, что индекс – это специальным образом организованный файл, отображающий содержимое той или иной таблицы, с помощью которого поиск данных по значениям в заданных столбцах происходит гораздо быстрее, чем без использования индекса. По существу использование индекса является экономичной заменой сортировки таблицы по заданным столбцам и последующего двоичного поиска. Создается индекс командой

CREATE [UNIQUE] INDEX <имя> ON <объект>
(<столбец> [DESC] [,…])

Здесь объект – это таблица или представление, а перечень столбцов указывает, по каким столбцам будет вестись отбор данных. Ключевое слово UNIQUE означает, что в индекс включается только одна строка из всех, имеющих одинаковые значения в указанных столбцах. DESC означает, что сортировка в этом столбце ведется по убыванию значений (по умолчанию она ведется в порядке их возрастания).

Интересно, что, несмотря на наличие команды создания индекса, в SQL отсутствуют команды, в которых индекс явно используется. Дело в том, что при отсутствии индексов СУБД автоматически создает их временные экземпляры при выполнении операторов SELECT, содержащих фразы ORDER BY, GROUP BY, а также во многих других случаях. После выполнения команды SELECT созданные индексы удаляются. Понятно, что если эта операция производится часто, то производительность системы уменьшается. Если же требуемый индекс уже существует как постоянный, то система использует его, не создавая и не удаляя

временный индекс. Узнать, какие индексы используются системой чаще всего, можно, использовав системную утилиту мониторинга (свою в каждой СУБД), и по ее отчету определить, какие индексы должны быть созданы на постоянной основе.

Следует дополнительно отметить, что наличие постоянных индексов в целом замедляет работу системы, так как при добавлении или удалении строк таблицы изменения должны быть внесены и во все связанные с таблицей индексы. Кроме того, при интенсивной модификации таблицы (вставке – удалении строк) индексы «портятся» – перестают быть сбалансированными, что уменьшает эффективность их использования. Для восстановления балансировки индекс должен быть построен заново, что может быть, в частности, сделано его удалением и повторным созданием. Поэтому еще раз стоит подчеркнуть, что вопрос создания и использования индексов весьма творческий и окончательно может быть решен только в процессе эксплуатации системы.

Удаляется индекс командой

DROP INDEX <имя> ON < объект > .

Команды модификации индекса существуют не во всех СУБД, а в тех, где существуют, значительно отличаются друг от друга. В качестве примера такой команды можно привести простейший вариант этой команды в SQL Server:

ALTER INDEX <имя> ON <объект> REBUILD

Эта команда по существу эквивалентна удалению и повторному созданию индекса с тем же описанием.

3.10. Хранимые процедуры и триггеры

При разработке крупных информационных систем типична ситуация, когда различные модули этой системы используют одни и те же типовые процедуры, например расчет средней зарплаты по подразделению. В этом случае целесообразно перенести эти процедуры с клиентской части системы на серверную. Это позволит снизить нагрузку как на клиентскую часть, так и на компьютерную сеть, поскольку обмен между клиентом и сервером уменьшится. Кроме того, это позволит централизовать модификацию совместно используемых модулей, так как они будут находиться в одном месте – на сервере.

Перенос логики приложений на сервер имеет и отрицательные стороны. Во-первых, увеличивается нагрузка на сервер,

что при большом числе запросов может оказаться критичным. Во-вторых, логика приложений становится менее прозрачной и при несогласованной модификации клиентской и серверной составляющих могут возникнуть ошибки, выявление которых будет представлять большие сложности. Наконец, надо отметить, что языки программирования, на которых пишутся серверные программы, являются системно зависимыми, причем весьма несовместимыми. Это в значительной степени затрудняет перевод информационной системы на другую платформу, если такой переход предполагается когда-либо в будущем.

Важной разновидностью хранимых процедур являются триггеры – процедуры, которые автоматически выполняются до, после или вместо определенных событий, к которым относятся вставка, удаление и замена строк в таблицах базы данных. Основным назначением триггеров является проверка названных операций на корректность, а также дополнительная обработка данных, например пересчет итоговых сумм, изменившихся в результате модификации таблицы. Языком программирования триггеров служит тот же язык, что и для хранимых процедур.

Примером оператора создания триггера в СУБД MySQL 5.5 может служить следующий текст:

```
delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|
delimiter ;
```

Не вдаваясь в специфику реализации языка SQL в этой СУБД, отметим, что перед выполнением команды создания триггера, текст выполняемой части которого (тела триггера) составляет три строки, производится замена ограничителя строки (delimiter) для того, чтобы эти строки не выполнялись немедленно, а только записывались в базу данных.

3.11. Многопользовательские возможности SQL

Как мы уже отмечали, основным предназначением баз данных является коллективная обработка хранимой информации. При этом различным категориям пользователей требуются разные ее подмножества и разные возможности по обработке этой информации. Язык SQL содержит типовые средства для реализации такого разграничения пользователей и их прав по отношению к обработке информации.

Ключевыми понятиями здесь являются пользователь, объекты и права (или привилегии) по отношению к этим объектам. Можно сказать, что с точки зрения системы пользователь – это совокупность прав по отношению ко всевозможным объектам системы.

Обычно для создания пользователя в системе используется команда CREATE USER, которая назначает пользователю имя и дополнительные характеристики (пароль, начальные права и т. д.). В дальнейшем права могут назначаться и изыматься с помощью команд GRANT (назначение) и REVOKE (изъятие). Формат этих команд следующий:

GRANT <права> ON <объект>
 TO {<имя пользователя> | PUBLIC}
REVOKE <права> ON <объект>
 FROM {<имя пользователя> | PUBLIC}

Мы рассмотрим простейший вариант этих команд, когда в качестве объектов могут выступать либо реальные, либо виртуальные таблицы (представления). В этом случае пользователю могут быть в любой комбинации предоставлены (назначены) права на просмотр таблиц (SELECT), добавление строк (INSERT), удаление строк (DELETE) и изменение строк (UPDATE). Соответствующие ключевые слова перечисляются через запятую после имени оператора назначения или изъятия прав. Если набор прав включает все четыре возможности, можно использовать вариант ALL PRIVILEGES. При этом права могут предоставляться двумя способами: либо безадресно (всем пользователям) – с использованием ключевого слова PUBLIC, либо с указанием имени конкретного пользователя. При этом указанные способы действуют независимо: если пользователю были персонально назначены какие-либо права, а затем выдана команда на изъятие таких прав,

назначенных безадресно, то права пользователя, персонально ему назначенные, у него останутся.

Приведем примеры использования команд GRANT и REVOKE.

```
GRANT SELECT,INSERT,DELETE,UPDATE ON Table1 TO PUBLIC
GRANT ALL PRIVILEGES ON Table1 TO Ivanov
REVOKE DELETE,UPDATE ON Table1 FROM Ivanov
REVOKE INSERT,DELETE,UPDATE ON Table1 FROM PUBLIC
```

Первая команда разрешает любому пользователю выполнять все возможные операции с таблицей Table1. Вторая делает то же самое, но уже применительно к конкретному пользователю Ivanov. Третья команда оставляет этому пользователю права только на добавление строк и просмотр таблицы. Наконец, четвертая команда оставляет всем пользователям лишь возможность просмотра таблицы, однако право на добавление строк у конкретного пользователя Ivanov остается.

Отдельно следует остановиться на управлении правами доступа применительно к представлениям. Наиболее интересен здесь вариант обновляемого представления. В этом случае можно предоставить пользователю право, например, просмотра всей базовой таблицы, а для представления, охватывающего лишь часть столбцов и строк, относящихся к этому пользователю, предоставить ему право изменения содержащейся в этой части информации. Тем самым мы получаем возможность управления правами пользователей по отношению не только к целым таблицам, но и к их частям.

В заключение рассмотрим наиболее часто используемое расширение команд управления пользователями. Речь идет о групповом управлении правами. Если количество обслуживаемых системой пользователей велико, то обычно их можно разбить на группы, которым следует назначить одинаковые права. В этом случае в систему управления правами вводится еще одно понятие – группы пользователей (в другой терминологии – роли), и действия по назначению-изъятию прав могут производиться применительно к группе. Для предоставления пользователю прав, которые назначены группе, достаточно включить пользователя в эту группу. Тем самым появляется третий способ назначения прав, причем все три способа действуют независимо друг от друга, так что пользователь может иметь один набор прав, предоставленный ему как неопределенному (PUBLIC) пользователю,

другой – назначенный ему персонально и третий – полученный им в составе группы. Описанный механизм значительно повышает гибкость системы управления правами пользователей.

Контрольные вопросы

1. Как соотносятся стандарт языка SQL и его реализации?
2. Назовите основные группы команд языка SQL.
3. Можно ли считать язык SQL и его реализации языками программирования?
4. Какая команда SQL выводит на экран полное содержимое таблицы test_table?
5. С помощью какой конструкции SQL можно обеспечить вывод символьных данных, удовлетворяющих заданному шаблону?
6. Для чего используются внешние соединения?
7. В каких случаях необходимо использование вложенных запросов?
8. Какие ключи применяются при описании таблиц?
9. Что такое триггер?
10. Какие команды SQL используются для управления правами пользователей?

Практическое задание

Пусть stud – таблица со структурой, описанной в этой главе и содержащая сведения обо всех студентах факультета. Используя какую-либо СУБД, постройте оператор SQL, выводящий таблицу сведений о назначенной стипендии следующей структуры:

Группа	Фамилия	Имя	Отчество	Стипендия
ИВТ-11БО				
	Алексеев	Владимир	Сергеевич	1 200
	Борисов	Сергей	Степанович	
	Владимиров	Алексей	Ильич	2 000
	…	…	…	…
Итого				15 300
ИВТ-12БО				
…	…	…	…	…
Всего				426 300

4. Проектирование на основе принципов нормализации

4.1. Уровни моделирования базы данных

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

- реляционная модель данных – удобный способ представления данных предметной области;
- 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 небольшое число отношений с большим количеством атрибутов в каждом 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 главе 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:

1. Сама предметная область.
2. Модель предметной области.
3. Логическая модель данных.
4. Физическая модель данных.
5. 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 не значащие данные. Так, если в качестве предметной области выбрать учет товаров на складе, то понятия «накладная» и «счет-фактура» являются существенно важными понятиями, а то, что сотрудница, принимающая накладные, имеет двоих детей – это для учета 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 нем 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 18 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*).

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 учета 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 еще 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 опять-таки решения, принятые на уровне логического моделирования, определяют некоторые границы, в пределах которых можно развивать физическую мо-

дель 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 дополнительно 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, объем оперативной памяти, дисковые подсистемы и т. п. Очень большое значение имеет также настройка СУБД в пределах выбранной программно-аппаратной платформы.

Но 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.*

4.2. Функциональные зависимости. Правила вывода

Концепция функциональной зависимости между атрибутами отношения является фундаментальной, поскольку она лежит в основе теории проектирования баз данных. Приведем основные положения этой концепции.

Определение 4.1. Пусть R – отношение со схемой $R\{A_1, A_2, \ldots, A_n\}$. Набор атрибутов $Y \subset \{A_1, A_2, \ldots, A_n\}$ ***функционально зависит*** от набора $X \subset \{A_1, A_2, \ldots, A_n\}$ тогда и только тогда, когда

для любого допустимого значения переменной R из того, что два кортежа $t_1, t_2 \in R$ совпадают по значению X ($t_1.X = t_2.X$), следует, что эти кортежи совпадают по значению Y ($t_1.Y = t_2.Y$). Символически функциональная зависимость записывается как $X \to Y$.

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

Определение 4.2. Функциональная зависимость атрибута Y от составного атрибута X называется *полной*, если для любого подмножества $X' \subset X$ функциональная зависимость $X' \to Y$ не выполняется.

Полную функциональную зависимость будем обозначать как $X \xrightarrow{\text{п}} Y$. В этом случае атрибут X называется *детерминантом* зависимости.

Приведем пример функциональных зависимостей между атрибутами. Пусть имеется отношение СТУДЕНТ-ОЦЕНКИ со схемой

СТУДЕНТ-ОЦЕНКИ{*номер_ст, фам_ст, группа, тел, код_дисц, имя_дисц, час, оценка*},

где *ном_ст* – номер студенческого билета, *фам_ст* – фамилия студента, *тел* – контактный телефон, *код_ дисц, имя_дисц, час* – код, название дисциплины и число часов соответственно, *оценка* – оценка на экзамене по данной дисциплине. Первичным ключом отношения является составной атрибут

$$K = \{номер\_ст, код\_дисц\}.$$

Функциональные зависимости

Зависимость неключевых атрибутов от ключа отношения:

$номер\_ст, код\_дисц \to фам\_ст$

$номер\_ст, код\_дисц \to группа$

$номер\_ст, код\_дисц \to конт\_тел$

$номер\_ст, код\_дисц \to час$

$номер\_ст, код\_дисц \to имя\_дисц$

$номер\_ст, код\_дисц \xrightarrow{п} оценка$.

Зависимость атрибутов, характеризующих студента, от номера студенческого билета

$номер\_ст \to фам\_ст$

$номер\_ст \to группа$

$номер\_ст \to конт\_тел$.

Зависимость атрибутов, характеризующих дисциплину, от кода дисциплины

$код\_дисц \rightarrow час$

$код\_дисц \rightarrow имя\_дисц$.

Приведенные функциональные зависимости *не выводятся* из конкретного заполнения таблицы. Эти зависимости отражают взаимосвязи, обнаруженные между объектами предметной области. Они возникают, когда по значениям одних данных в предметной области можно определить значения других данных. Например, зная номер студенческого билета студента, по отношению СТУДЕНТ-ОЦЕНКИ можно определить его фамилию и группу, по коду дисциплины можно определить ее название и количество учебных часов по данной дисциплине.

Каждая функциональная зависимость *задает дополнительное ограничение* на данные, которые могут храниться в отношениях, т. е. задает правило целостности данных. Для корректности базы данных при выполнении операций модификации базы данных необходимо, вообще говоря, проверять все ограничения, определенные функциональными зависимостями.

Для каждого отношения существует вполне определенное множество функциональных зависимостей между атрибутами. Причем из одной или более функциональных зависимостей, присущих рассматриваемому отношению, можно логически вывести другие функциональные зависимости, также присущие этому отношению или, наоборот, установить, что некоторые зависимости являются следствиями других.

Полный набор функциональных зависимостей, выполняющихся для всех допустимых значений отношения, может быть очень большим. В этом случае проверка каждой зависимости как ограничения целостности данных приведет к недопустимо медленной работе с базой данных. Поэтому при разработке приложений необходимо предварительно удалить те зависимости, которые *логически выводятся* из других, т. е. являются их следствиями. У. Армстронг отобрал некоторые из свойств функциональных зависимостей, которые оказались удобными для решения этой задачи. Эти свойства получили название *правил* или *аксиом* Армстронга. В дальнейшем по практическим соображениям эти аксиомы были дополнены еще несколькими аксиомами. Совокупный набор аксиом вывода приведен в табл. 4.1.

Таблица 4.1

Аксиомы вывода функциональных зависимостей

1	Если $X' \subseteq X$	то $X \to X'$	рефлексивность
2	Если $X \to Y$	то $XZ \to Y$	приращение детерминанта
3	Если $X \to Y$ и $X \to Z$	то $X \to YZ$	объединение
4	Если $X \to Y$ и $Z \subseteq Y$	то $X \to Z$	декомпозиция
5	Если $X \to Y$ и $Z \to W$	то $XZ \to YW$	композиция
6	Если $X \to Y$ и $Y \to Z$	то $X \to Z$	транзитивность
7	Если $X \to Y$ и $YZ \to W$	то $XZ \to W$	псевдотранзитивность

Можно доказать, что эта система аксиом *полна* и *совершенна* в следующем смысле: во-первых, для данного множества Ф функциональных зависимостей любая зависимость, потенциально выводимая из Ф, может быть выведена на основе аксиом из таблицы 4.1, и, во-вторых, применение этих аксиом не может привести к выводу лишней зависимости.

Приведем пример использования аксиом вывода для сокращения исходного множества функциональных зависимостей. Пусть для отношения $R\{A,B,C,D,E,F\}$ выявлен следующий набор функциональных зависимостей между атрибутами:

$$\Phi = \{A \to B,\ A \to C,\ B \to C,\ BD \to E,\ ADF \to E\}.$$

Тогда путем формального сопоставления этих зависимостей с аксиомами Армстронга набор Φ можно свести к следующему набору $\Phi' \subset \Phi$:

$$\Phi' = \{A \to B,\ B \to C,\ BD \to E\}.$$

Для этого надо показать, что зависимости $A \to C$ и $ADF \to E$ выводятся из остальных. Схема вывода следующая:

1. Зависимость $A \to C$ есть следствие $A \to B$ и $B \to C$ по аксиоме транзитивности (6);

2. Зависимость $ADF \to E$ выводится за два шага: из $A \to B$ и $BD \to E$ в силу аксиомы псевдотранзитивности (7) выводится зависимость $AD \to E$. Добавим в левую часть этой зависимости атрибут F и по аксиоме приращения детерминанта (2) получим $ADF \to E$.

Таким образом, зависимости $A \to C$ и $ADF \to E$ являются логическими следствиями других зависимостей и могут не учитываться как ограничения целостности.

Заметим, что в данном примере атрибут F оказался как бы «лишним»: он не зависит ни от какой-либо комбинации других атрибутов и от него никакой атрибут не зависит. Проектировщику надо пересмотреть схему данного отношения и решить вопрос о целесообразности включения в него этого атрибута.

4.3. Декомпозиция отношений. Теорема Хита

При проектировании базы данных разработчик часто стремится поместить все атрибуты, описывающие объекты предметной области, в небольшое число отношений (в идеале – в одно отношение). Однако, как правило, такой проект оказывается плохим. Так, например, схема базы данных для хранения информации о студентах и их успеваемости в виде единственного отношения СТУДЕНТ-ОЦЕНКИ явно неудовлетворительна. Проблемы, которые могут возникнуть при работе с этой базой данных, следующие.

1. *Аномалия обновления* (UPDATE). В таблице присутствует значительная избыточность данных: фамилия, группа, контактный телефон одного и того же студента в такой таблице повторяются столько раз, сколько он сдавал экзамены, название и число учебных часов дисциплины повторяется столько раз, сколько студентов сдавало экзамен по этой дисциплине. При редактировании данных (например, изменение номера контактного телефона или числа учебных часов по дисциплине) пользователю вручную придется одно и то же исправление сделать в большом числе записей, при этом вероятность внесения ошибки будет достаточно высокой.

Причина аномалии – хранение в одном отношении разнородной информации (и о студентах, и об учебных дисциплинах, и о результатах экзаменов).

2. *Аномалия включения* (INSERT). В таблицу нельзя включить данные о студенте, пока он не сдал хотя бы один экзамен по какой-либо дисциплине, т. к. ключевой атрибут *код_дисц* обязательно должен иметь конкретное значение (не null!).

3. *Аномалия удаления* (DELETE). Если удалить из таблицы запись о студенте, сдавшем лишь *один* экзамен, то будет потеряна вся информация о нем (фамилия, номер телефона), которая может потребоваться в дальнейшем.

Эти проблемы (аномалии) можно устранить, если разбить отношение СТУДЕНТ-ОЦЕНКИ на три отношения со схемами

СТУДЕНТ{*номер_ст, фам_ст, группа,, конт_тел*}

ДИСЦИПЛИНА{$_{код\_дисц}$, *имя_дисц, час*},

СЕССИЯ{*номер_ст, код_дисц, оценка*}·

Заметим, что каждое отношение является проекцией исходного отношения СТУДЕНТ-ОЦЕНКА на соответствующие атрибуты. Процедура разбиения отношения на два или более отношений с помощью операции проекции называется *декомпозицией.*

Очевидно, что при декомпозиции *не должны теряться атрибуты* отношения, т. е. эти проекции в совокупности должны содержать (возможно, с повторениями) *все* атрибуты исходного отношения. Но при декомпозиции также не должны потеряться и сами данные. Данные можно считать не потерянными в том случае, если возможна обратная операция – по отношениям-проекциям можно восстановить исходное отношение *в точности в прежнем виде*.

Определение 4.3. 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:

$$R_1 \text{ NATURAL JOIN } R_2 = R.$$

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

Пример. Пусть дано отношение R:

Таблица 4.2

Отношение R

ТабНомер	*Фамилия*	*Зарплата*
001	Иванов	20 000
002	Петров	15 000
003	Сидоров	20 000

В этом отношении ключом является атрибут *ТабНомер*, и, следовательно, выполняются функциональные зависимости *ТабНомер* $\rightarrow$ *Фамилия* и *ТабНомер* $\rightarrow$ *Зарплата*.

Рассмотрим первый вариант декомпозиции отношения R на два отношения $R_1 = R[$*ТабНомер*, *Зарплата*$]$ и $R_2 = R[$*Фамилия*, *Зарплата*$]$, представленных в табл. 4.3 и 4.3 соответственно.

Таблица 4.3

Отношение R_1

ТабНомер	*Зарплата*
001	20 000
002	15 000
003	20 000

Таблица 4.4

Отношение R_2

Фамилия	*Зарплата*
Иванов	20 000
Петров	15 000
Сидоров	20 000

Естественное соединение этих проекций, имеющих общий атрибут «З*арплата*», очевидно, будет следующим (склеиваются строки, для которых $R_1.Зарплата = R_2.Зарплата$) (табл. 4.5).

Таблица 4.5

Отношение R_1 *NATURAL JOIN* $R_2 \neq R$

ТабНомер	*Фамилия*	*Зарплата*
001	Иванов	20 000
001	Сидоров	20 000
002	Петров	15 000
003	Иванов	20 000
003	Сидоров	20 000

Данная декомпозиция не является декомпозицией без потерь, т. к. исходное отношение *не восстанавливается в точном виде* по проекциям – при соединении появились лишние кортежи (выделены серым цветом).

Рассмотрим другой вариант декомпозиции: $\tilde{R}_1 = R[ТабНомер, Фамилия]$, $\tilde{R}_2 = R[ТабНомер, зарплата]$ (табл. 4.6. и 4.7).

Таблица 4.6

Отношение $\tilde{R}_1$

ТабНомер	*Фамилия*
001	Иванов
002	Петров
003	Сидоров

Таблица 4.7

Отношение $\tilde{R}_2$

ТабНомер	*Зарплата*
001	20 000
002	15 000
003	20 000

По данным проекциям, имеющим общий атрибут *ТабНомер*, исходное отношение, как легко проверить, восстанавливается в точном виде.

В примере рассмотрено только одно конкретное состояние отношения *R*. Но правильность второго варианта декомпозиции

будет справедлива при *любых* состояниях R. Гарантией этого служит следующая.

Теорема Хита. Пусть для отношения $R = \{A, B, C\}$ (*A, B* и *C* могут быть составными атрибутами) справедлива функциональная зависимость $A \to B$. Тогда декомпозиция отношения R на два отношения $R_1 = R[A, B]$ и $R_2 = R[A, C]$ будет декомпозицией без потерь.

Доказательство. Необходимо доказать, что R_1 *NATURAL JOIN* $R_2 = R$ *для любого состояния* отношения *R*. В левой и правой частях равенства стоят *множества* кортежей, поэтому для доказательства достаточно доказать два включения для двух *множеств* кортежей: $R \subseteq R_1$ *NATURAL JOIN* R_2 и R_1 *NATURAL JOIN* $R_2 \subseteq R$.

Докажем первое включение. Возьмем произвольный кортеж $r = (a, b, c) \in R$. Покажем, что он включается также и в R_1 *NATURAL JOIN* R_2. По определению проекции кортежи $r_1 = (a, b) \in R_1$ и $r_2 = (a, c) \in R_2$. По определению естественного соединения кортежи r_1 и r_2, имеющие одинаковое значение a общего атрибута *A*, будут соединены в процессе естественного соединения в кортеж $(a, b, c) \in R_1$ *NATURAL JOIN* R_2. Таким образом, включение доказано.

Для доказательства обратного включения возьмем произвольный кортеж $(a, b, c) \in R_1$ *NATURAL JOIN* R_2. Докажем, что он входит также и в R. Из определения естественного соединения следует, что имеются кортежи $r_1 = (a, b) \in R_1$ и $r_2 = (a, c) \in R_2$. Т. к. $R_1 = R[A, B]$, то существует некоторое значение c_1, такое что кортеж $(a, b, c_1) \in R$. Аналогично, существует некоторое значение b_1, такое что кортеж $(a, b_1, c) \in R$. Кортежи (a, b, c_1) и (a, b_1, c) имеют одинаковое значение атрибута *A*, равное *a*. Из этого, в силу функциональной зависимости $A \to B$, следует, что $b = b_1$. Таким образом, кортеж $(a, b, c) \in R$. Обратное включение доказано.

В приведенном примере причина неправильной декомпозиции в первом варианте состоит в том, что при этой декомпозиции была *потеряна* функциональная зависимость $ТабНомер \to Зарплата$. Во втором варианте, удовлетворяющем условию теоремы Хита, эта зависимость сохранена в отношении $\tilde{R}_2$.

4.4. Нормальные формы

Излагаемый здесь метод проектирования базы данных основан на понятии *нормальных форм* реляционных отношений. Все шаги метода проектирования будут иллюстрироваться на типовом конкретном примере следующего вида [2, 3].

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 ровно один сотрудник.
6. Каждый сотрудник числится в одном отделе.
7. Каждый сотрудник имеет телефон, находящийся в отделе сотрудника.

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

О каждом сотруднике необходимо хранить табельный номер и фамилию. Табельный номер является уникальным для каждого сотрудника.

Каждый отдел имеет уникальный номер.

Каждый проект имеет номер и наименование. Номер проекта является уникальным.

Каждое задание проекта имеет номер, уникальный в пределах проекта. Задания в разных проектах могут иметь одинаковые номера.

4.4.1. Первая нормальная форма (1НФ) и аномалии отношения

Понятие первой нормальной формы уже обсуждалось в главе 2. Первая нормальная форма (1НФ) – это обычное отношение. Согласно определению (см. раздел 2.1), 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 со схемой:
СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ{*НомСотр*, *Фам, НомОтд, Тел, НомПро, Проект, НомЗадан*},

где

НомСотр – табельный номер сотрудника;

Фам – фамилия сотрудника;

НомОтд – номер отдела, в котором числится сотрудник;

Тел – телефон сотрудника;

НомПро – номер проекта, в котором работает сотрудник;

Проект – наименование проекта, в котором работает сотрудник;

НомЗадан – номер задания, над которым работает сотрудник.

Так как каждый сотрудник в каждом проекте выполняет ровно одно задание, то в качестве потенциального ключа отношения необходимо взять пару атрибутов {*НомСотр*, *НомПро*}.

В текущий момент состояние предметной области отражается следующими фактами:

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

Это состояние отражается в табл. 4.8 (курсивом выделены ключевые атрибуты).

В разделе 4.3. были рассмотрены проблемы, которые возникают при работе с неудачно спроектированным отношением – это т. н. аномалии обновления, вставки и удаления. Для отношения *СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ* все эти аномалии имеют место.

Таблица 4.8

Отношение СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ

НомСотр	*Фам*	*НомОтд*	*Тел*	*НомПро*	*Проект*	*НомЗадан*
001	Иванов	1	11-22-33	1	Альфа	1
001	Иванов	1	11-22-33	2	Бета	1
002	Петров	1	11-22-33	1	Альфа	2
003	Сидоров	2	33-22-11	1	Альфа	3
003	Сидоров	2	33-22-11	2	Бета	2

Аномалии обновления (UPDATE)

Фамилии сотрудников, наименования проектов, номера телефонов повторяются во многих кортежах отношения (избыточность данных). 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 специальную программу-триггер, которая при обновлении одной записи корректно исправляла бы данные и в других записях.

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, Сергеев, и он пока не участвует ни в одном проекте, то мы должны вставить в отношение кортеж (004, Сергеев, 2, 33-22-11, null, null, null). Это сделать невозможно, т. к. атрибут *НомПро* (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.

Аномалии удаления (DELETE)

При удалении некоторых данных может произойти потеря другой информации. Например, если закрыть проект «Альфа» и

удалить все строки, в которых он встречается, то будут потеряны все данные о сотруднике Петрове. Если удалить данные о сотруднике Сидорове, 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.

4.4.2. Вторая нормальная форма (2НФ)

Далее будем предполагать, что в реляционном отношении имеется только один потенциальный ключ (он же и первичный).

Определение 4.5. Отношение R находится во второй нормальной форме (2НФ) тогда и только тогда, когда отношение находится в 1НФ и каждый неключевой атрибут[2] функционально полно зависит от первичного ключа.

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

Отношение *СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ* не находится в 2НФ, т. к. есть атрибуты, зависящие от части составного ключа *НомСотр, НомПро*:

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

  НомСотр → Фам,
  НомСотр → НомОтд,
  НомСотр → Тел;

- зависимость наименования проекта от номера проекта является зависимостью от части сложного ключа:

  НомПро → 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.

[2] Неключевой атрибут – это атрибут, не входящий в состав никакого потенциального ключа.

Отношение *СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ* декомпозируем на три отношения: *СОТРУДНИКИ-ОТДЕЛЫ, ПРОЕКТЫ, ЗАДАНИЯ*.

1. Отношение *СОТРУДНИКИ-ОТДЕЛЫ* {*НомСотр, Фам, НомОтд, Тел*}: функциональные зависимости:

НомСотр → *Фам,*
НомСотр →*НомОтд,*
НомСотр →*Тел.*
НомОтд →*Тел.*

Таблица 4.9

Отношение СОТРУДНИКИ-ОТДЕЛЫ

НомСотр	*Фам*	*НомОтд*	*Тел*
001	Иванов	1	11-22-33
002	Петров	1	11-22-33
003	Сидоров	2	33-22-11

2. Отношение *ПРОЕКТЫ {НомПро, Проект}:* функциональные зависимости:

НомПро → *Проект.*

Таблица 4.10

Отношение ПРОЕКТЫ

НомПро	*Проект*
1	Альфа
2	Бета

3. Отношение ЗАДАНИЯ {*НомСотр, НомПро, НомЗадан*}: функциональные зависимости:

$\{НомСотр, НомПро\} \xrightarrow{полно} НомЗадан$

Таблица 4.11

Отношения ЗАДАНИЯ

НомСотр	*НомПро*	*НомЗадан*
001	1	1
001	2	1
002	1	2
003	1	3
003	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 составной ключ, но единственный неключевой атрибут *НомЗадан* функционально полно зависит от ключа $\{НомСотр, НомПро\}$.

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

В отношение СОТРУДНИКИ-ОТДЕЛЫ нельзя вставить кортеж (004, Сергеев, 1, 33-22-11), т. к. при этом получится, что два сотрудника из 1-го отдела (Иванов и Сергеев) 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 введен по двум причинам – по ошибке человека, вводящего данные о новом сотруднике, или потому что номер в отделе действительно изменился. Тогда можно написать программу-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 введен ошибочно), то кортеж с данными о новом сотруднике будет вставлен, но номер телефона у него будет 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, порожденная тем, что в одном отношении хранится разнородная информация (о сотрудниках и об отделах). Наличие этой аномалии порождает определенную сложность разработки базы данных. База данных, основанная на такой модели, будет работать правильно только при наличии дополнительного программного кода в виде триггеров.

Далее, 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.

Оставшиеся аномалии удаления

При удалении некоторых данных по-прежнему может произойти потеря другой информации. Например, если удалить данные о сотруднике Сидорове, 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).

Заметим, что при переходе к 2НФ некоторые аномалии исчезли. Остались трудности в разработке базы данных, связанные с необходимостью написания триггеров, поддерживающих целостность базы данных. Эти трудности теперь связаны только с одним отношением СОТРУДНИКИ-ОТДЕЛЫ.

4.4.3. Третья нормальная форма (3НФ)

Будем, как и ранее, предполагать, что в отношении существует лишь один потенциальный ключ.

Определение 4.6. Атрибуты называются взаимно независимыми, если ни один из них не является функционально зависимым от другого.

Определение 4.7. Отношение *R* находится в третьей нормальной форме (3НФ) тогда и только тогда, когда а) отношение находится в 2НФ и б) все неключевые атрибуты взаимно независимы.

Замечание. Иногда 3НФ определяют следующим образом. Отношение находится в 3НФ тогда и только тогда, когда а) оно находится во 2НФ и б) отсутствуют транзитивные зависимости

неключевых атрибутов A от потенциального ключа K[3]. Очевидно, что это определение 3НФ эквивалентно определению 4.7.

Отношение СОТРУДНИКИ-ОТДЕЛЫ не находится в 3НФ, т. к. имеется функциональная зависимость неключевых атрибутов (зависимость номера телефона от номера отдела): *НомОтд* $\rightarrow$ *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* $\{НомСотр, Фам, НомОтд\}$.

Функциональные зависимости:

НомСотр $\rightarrow$ *Фам,*

НомСотр $\rightarrow$ *НомОтд.*

Таблица 4.12

Отношение СОТРУДНИКИ

НомСотр	*Фам*	*НомОтд*
001	Иванов	1
002	Петров	1
003	Сидоров	2

Отношение *ОТДЕЛЫ* $\{НомОтд, Тел\}$.

Функциональные зависимости:

НомОтд $\rightarrow$ *Тел.*

Таблица 4.13

Отношение ОТДЕЛЫ

НомОтд	*Тел*
1	11-22-33
2	33-22-11

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

Таким образом, все обнаруженные аномалии обновления устранены. Реляционная модель, состоящая из четырех отношений

[3] Напомним, что зависимость $K \rightarrow A$ называется транзитивной, если существует атрибут B такой, что имеются функциональные зависимости $K \rightarrow B$ и $B \rightarrow A$, и отсутствует функциональная зависимость $B \rightarrow K$.

СОТРУДНИКИ, ОТДЕЛЫ, *ПРОЕКТЫ, ЗАДАНИЯ*, находящихся в 3НФ, 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)

Алгоритм приведения отношений к 3НФ описывается следующим образом.

Шаг 1 (приведение к 1НФ). На первом шаге задается 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 составного 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: $R\{K_1, K_2, A_1, \ldots, A_m, B_1, \ldots, B_n\}$. Ключ $\{K_1, K_2\}$.

Функциональные зависимости:

$\{K, K_2\} \to \{A_1, \ldots, A_m, B_1, \ldots, B_n\}$ – зависимость всех атрибутов от ключа отношения.

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

Декомпозированные отношения:

$R_1\{K_1, K_2, B_1, \ldots, B_n\}$ – остаток от исходного отношения, ключ $\{K_1, K_2\}$.

$R\{K_1, A_1, \ldots, A_m\}$ – 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 от других неключевых атрибутов, то проводим декомпозицию этих отношений следующим образом: те неключевые атрибуты, кото-

рые зависят от других неключевых атрибутов, выносятся в отдельное отношение. В новом отношении ключом становится детерминант функциональной зависимости:

Исходное отношение: $T\{K, A, B\}$, ключ K.

Функциональные зависимости:

$K \to \{A, B\}$ – зависимость всех атрибутов от ключа отношения.

$A \to B$ – зависимость некоторых неключевых атрибутов от других неключевых атрибутов.

Декомпозированные отношения:

$T_1\{K, A\}$ – остаток от исходного отношения, ключ K.

$T_2\{A, B\}$ – атрибуты, вынесенные из исходного отношения вместе с детерминантом функциональной зависимости, ключ A.

На практике при создании логической модели данных, как правило, не следуют прямо приведенному алгоритму нормализации. Опытные разработчики обычно сразу строят отношения в 3НФ. Кроме того, основным средством разработки логических моделей данных являются различные варианты *ER-диаграмм*, о которых пойдет речь в следующей главе. Особенность этих диаграмм в том, что они сразу позволяют создавать отношения в 3НФ. Тем не менее приведенный 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 чем-либо упомянуть, разработчик может неправильно понять эксперта, во время разработки могут измениться правила, принятые в предметной области, и т. д. Все 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.

4.4.4. Нормальная форма Бойса-Кодда (НФБК)

При приведении отношений при помощи алгоритма нормализации к отношениям в 3НФ предполагалось, что все отношения

содержат один потенциальный ключ. Это не всегда верно. Рассмотрим следующий пример отношения, содержащего два ключа.

Пример. Пусть на факультете организованы в течение учебного года консультации для студентов по различным дисциплинам. Пусть выполнены следующие условия:

1. Студент может консультироваться по нескольким дисциплинам.
2. Преподаватель-консультант консультирует только по одной дисциплине.
3. За каждой дисциплиной может быть закреплено несколько консультантов.
4. По выбранной дисциплине студент может консультироваться только у одного преподавателя.

Пример представления данных о консультациях приведен в табл. 4.14.

Таблица 4.14

Отношение КОНСУЛЬТАЦИИ

студент	*дисциплина*	*консультант*
Иванов	мат_анализ	Г. В. Шабаршина
Петров	мат_анализ	В. А. Бондаренко
Петров	базы_данных	А. В. Зафиевский
Сидоров	мат_анализ	В. А. Бондаренко
Сидоров	базы_данных	А. В. Зафиевский

Выявим функциональные зависимости для отношения *КОНСУЛЬТАЦИИ*:

1) {студент, дисциплина} → консультант.

2) {студент, консультант} → дисциплина.

3) консультант → дисциплина.

В этом отношении два потенциальных ключа K_1= {*студент, дисциплина*} и K_2={*студент, консультант*}. При этом у ключей K_1 и K_2 имеется общий атрибут – *студент.* Отметим, что отношение **не содержит** неключевых атрибутов, зависящих от части составного ключа (см. определение 2НФ). Действительно, от части ключа K_2 зависит атрибут *дисциплина*, но он сам является ключевым. Таким образом, отношение формально находится в 2НФ. Кроме того, в отношении нет зависимостей между неключевыми атрибутами (*консультант* входят в K_1, а *дисциплина* – *в* K_2). Следова-

тельно, отношение *КОНСУЛЬТАЦИИ* находится в 3НФ. Тем не менее при работе с этой таблицей возможны следующие аномалии:

- **аномалия включения** – нельзя добавить в таблицу нового преподавателя-консультанта, пока к нему не запишется на консультацию хотя бы один студент;
- **аномалия удаления** – если студент Иванов откажется от консультаций по математическому анализу, то соответствующая строка удаляется из таблицы и при этом теряется информация о том, что преподаватель Г. В. Шабаршина является консультантом по математическому анализу.

Определение 4.8. Отношение *R* находится в нормальной форме Бойса-Кодда (НФБК) тогда и только тогда, когда детерминанты всех полных функциональных зависимостей, присутствующих в отношении *R*, являются потенциальными ключами.

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

НФБК является более строгой версией 3НФ. Иными словами, любое отношение, находящееся в НФБК, находится в 3НФ. Обратное неверно, что показывает приведенный выше пример.

Задача. Нормализуйте отношение *КОНСУЛЬТАЦИИ* декомпозицией на два отношения так, чтобы для каждого из них выполнялись условия НФБК.

4.4.5. Четвертая нормальная форма (4НФ)

Предположим, мы хотим хранить следующую информацию о сотрудниках: фамилия сотрудника, проекты, в которых он работает, имена его детей. Сотрудник Иванов работает в двух проектах и у него двое детей; у Петрова, работающего только в одном проекте, один ребенок. Если всю эту информацию разместить в одной таблице, то она будет иметь, например, такой вид:

СОТРУДНИКИ-ПРОЕКТЫ-ДЕТИ

сотрудник	*проект*	*дети*
Иванов	Альфа	Саша
Иванов	Альфа	Оля
Иванов	Бета	Саша
Иванов	Бета	Оля
Петров	Альфа	Аня

Отношение *СОТРУДНИКИ-ПРОЕКТЫ-ДЕТИ* явно неудачно спроектировано. Например, если у Иванова появляется еще один ре-

бенок, то новых записей появится столько, в скольких проектах он участвует. Если Иванов привлекается к работе еще в одном проекте, то придется просмотреть атрибут *дети* и внести столько новых записей, сколько у Иванова детей.

Декомпозиция отношения *СОТРУДНИКИ-ПРОЕКТЫ-ДЕТИ* для устранения указанных аномалий не может быть выполнена на основе функциональных зависимостей, т. к. это отношение *не содержит никаких функциональных зависимостей*. Это отношение является полностью ключевым, т. е. ключом отношения является все множество атрибутов. Но ясно, что какая-то взаимосвязь между атрибутами имеется. Эта взаимосвязь описывается понятием ***многозначной зависимости***.

Определение 4.9. Пусть R – отношение и A, B, C – некоторые из его атрибутов (или *непересекающиеся* множества атрибутов). Тогда атрибуты (множества атрибутов) B и C ***многозначно зависят*** от A (обозначается $A \to\to B \mid C$), если для каждого значения A имеется набор значений атрибута B и набор значений атрибута C. Однако входящие в эти наборы значения атрибутов B и C не зависят друг от друга.

Очевидно, что каждая функциональная зависимость является частным случаем многозначной, но не каждая многозначная зависимость является функциональной.

Определение 4.10. Многозначная зависимость $A \to\to B \mid C$ называется нетривиальной многозначной зависимостью, если *не существует функциональных зависимостей* $A \to B$ и $A \to C$.

В примере многозначная зависимость $сотрудник \to\to проект \mid дети$ очевидно является нетривиальной.

Определение 4.11. Реляционное отношение находится в 4НФ, если она находится в НФБК и в нем отсутствуют многозначные нетривиальные зависимости.

Нормализация отношения R с присутствующими в нем многозначными зависимостями осуществляется путем декомпозиции на два отношения, каждое из которых является проекцией R на соответствующие атрибуты. Теоретическим обоснованием такой декомпозиции является следующая

Теорема Фейджина. Пусть имеется отношение R с атрибутами A, B, C (в общем случае, составными). Отношение R

декомпозируется без потерь на проекции $R1 = R[A,B]$ и $R2 = R[A,C]$ тогда и только тогда, когда для него выполняется $A \to\to B \mid C$.

Теорема Фейджина является обобщением теоремы Хита (см. раздел 4.3) на случай многозначных зависимостей.

Декомпозиция указанной выше таблицы выглядит следующим образом:

СОТРУДНИКИ-ПРОЕКТЫ

сотрудник	*проект*
Иванов	Альфа
Иванов	Бета
Петров	Альфа

СОТРУДНИКИ-ДЕТИ

сотрудник	*дети*
Иванов	Саша
Иванов	Оля
Петров	Аня

В силу теоремы Фейджина это будет декомпозиция без потерь.

Следует отметить, что существует еще одна нормальная форма реляционных отношений – так называемая пятая нормальная форма (5НФ). 5НФ – это последняя нормальная форма, которую можно получить путем декомпозиции. Ее условия достаточно нетривиальны, и на практике 5НФ используется крайне редко. Достаточно подробно 5НФ описана в [3].

Таким образом, мы пришли к итоговой схеме процедуры нормализации отношений:

1. Отношение в 1НФ необходимо разбить на проекции для исключения всех неполных функциональных зависимостей от единственного ключа. Как результат, в итоге будет получен набор отношений в 2НФ.

2. Отношение, находящееся в 2НФ, следует разбить на проекции для исключения функциональных зависимостей между неключевыми атрибутами. В результате будет получен набор отношений в 3НФ.

3. Полученные отношения в 3НФ следует разбить на проекции для исключения любых функциональных зависимостей, в которых детерминанты не являются потенциальными ключами. В результате такого приведения будет получен набор отношений в НФБК.

4. Отношения, находящиеся в НФБК, необходимо разбить на проекции для исключения любых многозначных зависимостей, которые не являются функциональными. В итоге получим набор отношений в 4НФ.

Сравнение нормализованных и ненормализованных логических моделей по влиянию на производительность базы данных показывает, что более сильно нормализованные отношения оказываются лучше спроектированы. Они больше соответствуют предметной области, легче в разработке, для них быстрее выполняются операции модификации базы данных. Правда, это достигается ценой некоторого замедления выполнения операций выборки данных.

У слабо нормализованных отношений единственное преимущество – если к базе данных обращаться только с запросами на выборку данных, то для слабо нормализованных отношений такие запросы выполняются быстрее. Это связано с тем, что в таких отношениях уже как бы произведено соединение отношений и на это не тратится время при выборке данных.

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

4.5. 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 учета, 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 счета N_1, добавить эту сумму на счет N_2». 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 счета N_1, но не поступили на счет N_2). 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, таким образом, известна заранее еще на этапе проектирования системы. Таким образом, критическим для OLTP-приложений являются скорость и надежность выполнения коротких операций обновления данных. Чем выше уровень нормализации данных в OLTP-приложении, тем оно, как правило, быстрее и надежнее. Отступления от этого правила могут происходить тогда, когда уже на этапе разработки известны некоторые часто возникающие запросы, требующие соединения отношений 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 обобщенный 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 объем 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 объемы 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 объемы 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 построен 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 не страшны, т. к. обновление происходит только в момент загрузки новой порции данных. При этом происходит как добавление новых данных, так и пересчет итогов.

Контрольные вопросы

1. Рассмотрим схему отношения $R(A,B,C,D)$ и следующий набор ФЗ:

$A\,B \to C$, $C \to D$ и $D \to A$.

а) Каковы все нетривиальные ФЗ, которые следуют из заданных ФЗ? Ограничьтесь рассмотрением ФЗ с единственным атрибутом в правой части.

б) Каковы все ключи отношения R?

2. Что означает фраза «декомпозиция отношения R на два отношения $R1$ и $R2$ является декомпозицией без потерь»?

3. Для отношения $R(A,B,C,D)$ существует функциональная зависимость $AC \to D$. Будет ли декомпозиция R на два отношения $S = R[A,B,C]$ и $T = R[A,C,D]$ декомпозицией без потерь?

4. Пусть имеется отношение $R(A,B,C)$ с многозначной зависимостью $A \to\to B \mid C$. Если известно, что в текущем экземпляре отношения содержатся кортежи (a,b_1,c_1), (a,b_2,c_2) и (a,b_3,c_3), какие другие кортежи также должны присутствовать в том же экземпляре R?

Практическое задание

Предположим, что дано отношение, предназначенное для хранения информации о человеке, включающей его имя, номер полиса медицинского страхования, дату рождения, те же сведения для каждого из детей этого человека, а также данные о номере и марке каждого автомобиля, которым человек владеет.

Кортеж отношения выглядит как (*n, p, b, cn, cp, cb, as, am*), где *n* – имя человека, *p* – номер полиса медицинского страхования, *b* – дата рождения, *cn* – имя ребенка, *cp* – номер полиса медицинского страхования ребенка, *cb* – дата рождения ребенка, *as* – номер автомобиля, *am* – марка автомобиля.

Для заданного отношения:

- укажите функциональные и многозначные зависимости, которым, по вашему мнению, оно должно отвечать;
- предложите вариант декомпозиции в 4НФ.

5. Логическое моделирование. Модель «сущность-связь»

Моделирование структуры базы данных при помощи алгоритма нормализации, описанного в главе 4, имеет серьезные недостатки:

1. Первоначальное размещение всех атрибутов в одном отношении является очень неестественной операцией. Интуитивно разработчик сразу проектирует несколько отношений в соответствии с обнаруженными сущностями. Даже если совершить насилие над собой и создать одно или несколько отношений, включив в них все предполагаемые атрибуты, то совершенно неясен смысл полученного отношения.

2. Невозможно сразу определить полный список атрибутов. Пользователи имеют привычку называть разными именами одни и те же вещи или, наоборот, называть одними именами разные вещи.

3. Для проведения процедуры нормализации необходимо выделить зависимости атрибутов, что тоже очень нелегко, т. к. необходимо *явно выписать все зависимости*, даже те, которые являются очевидными.

В реальном проектировании структуры базы данных применяется другой метод – так называемое ***семантическое моделирование***. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты ***диаграмм сущность-связь*** или ER-диаграмм (ER – Entity-Relationship).

Первый вариант модели сущность-связь был предложен в 1976 г. П. Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. По сути, все варианты ER-диаграмм исходят из одной идеи – рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов) и взаимосвязей между сущностями.

Мы опишем работу с ER-диаграммами близко к нотации Баркера как довольно легкой в понимании основных идей. Данная

глава является скорее иллюстрацией методов семантического моделирования, чем полноценным введением в эту область.

5.1. Основные понятия ER-диаграмм

Определение 5.1. Сущность – это класс однотипных объектов, информация о которых должна быть учтена в модели.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. Примерами сущностей могут быть такие классы объектов, как «Поставщик», «Сотрудник», «Накладная», «Студент», «Преподаватель», «Дисциплина». Каждая сущность в модели изображается в виде прямоугольника с наименованием:

Сотрудник

Определение 5.2. Экземпляр сущности – это конкретный представитель данной сущности.

Например, представителем сущности «Сотрудник» может быть «Сотрудник Иванов». Экземпляры сущностей должны быть *различимы*, т. е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

Определение 5.3. Атрибут сущности – это именованная характеристика, являющаяся некоторым свойством сущности.

Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности «Сотрудник» могут быть такие атрибуты, как «Табельный номер», «Фамилия», «Имя», «Отчество», «Должность», «Зарплата» и т. п.

Сотрудник
Табельный номер
Фамилия
Имя
Отчество

Рис. 5.1

Атрибуты изображаются в пределах прямоугольника, определяющего сущность (рис. 5.1).

Определение 5. 4. Ключ сущности – это *минимальный* набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности. Минимальность заключается в том, что удаление любого атрибута из ключа нарушает его уникальность.

Сущность может иметь несколько различных ключей. Ключевые атрибуты изображаются на диаграмме подчеркиванием, как показано на рис. 5.2.

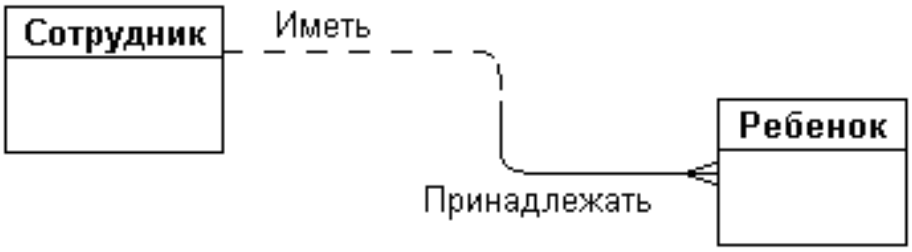


Рис. 5.2. Связь между двумя сущностями

Определение 5.5. Связь – это некоторая ассоциация между *двумя* сущностями. Одна сущность может быть связана с другой сущностью или сама с собою.

Связи позволяют по одной сущности находить другие сущности, связанные с нею. Например, связи между сущностями могут выражаться следующими фразами – «СОТРУДНИК может иметь несколько ДЕТЕЙ», «каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ».

Графически связь изображается линией, соединяющей две сущности (рис. 5.2):

Каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: «иметь», «принадлежать» и т.п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Каждая связь может иметь один из следующих ***типов связи*** (рис. 5.3).

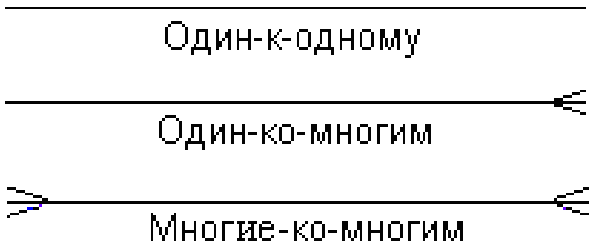


Рис. 5.3. Типы связей

Связь типа ***один-к-одному*** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй

сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа ***один-ко-многим*** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны «один») называется ***главной*** в этой связи, правая (со стороны «много») – ***подчиненной***. Характерный пример такой связи приведен на рис. 5.2.

Связь типа ***много-ко-многим*** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Пример такой связи приведен на рис. 5.4 (один студент сдает экзамены по многим предметам, по одному предмету экзаменуется много студентов).

Рис. 5.4. Пример связи «многие-ко-многим»

Тип связи много-ко-многим является *временным* типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности подчиненного типа для главных сущностей. Атрибутом этой промежуточной сущности является характеристика (числовая или символьная или другого типа) исходной связи. Пример преобразования связи «многие-ко-многим» в примере на рис. 5.4 показан ниже (рис. 5.5).

Рис. 5.5. Преобразованная связь «экзамен»

Каждая связь может иметь одну из двух ***модальностей связи***:

– – – – – – – Может

——————— Должен

Модальность «***может***» означает, что экземпляр одной сущности *может быть связан* с одним или несколькими экземплярами другой сущности, *а может быть и не связан* ни с одним экземпляром. Связь такого типа иногда называется *необязательной*.

Модальность «***должен***» означает, что экземпляр одной сущности *обязан быть связан не менее чем с одним* экземпляром другой сущности (*обязательная* связь).

Связь может иметь *разную модальность* с разных концов как на рис. 5.2.

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

<Каждый экземпляр СУЩНОСТИ 1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ> <ТИП СВЯЗИ> <экземпляр СУЩНОСТИ 2>.

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рис. 5.2 читается так:

слева направо: «каждый сотрудник может иметь несколько детей»;

справа налево: «Каждый ребенок обязан принадлежать ровно одному сотруднику».

(Здесь предполагается, что среди сотрудников нет семейных пар, у которых есть дети, в противном случае это была бы связь «многие-ко-многим»).

Отметим, что между двумя сущностями может быть задано несколько связей с различными смысловыми нагрузками, как, например, показано на рис. 5.6.

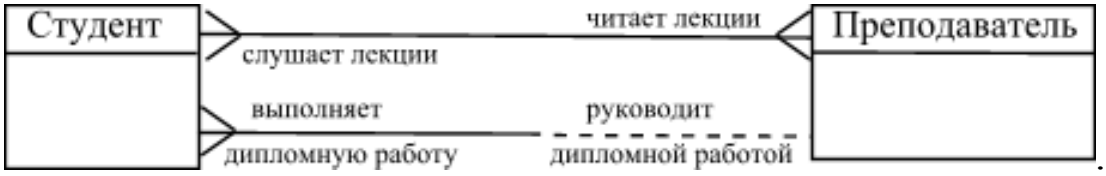


Рис. 5.6. Возможные связи между двумя сущностями

5.2. Пример разработки простой ER-модели

При разработке ER-моделей мы должны получить следующую информацию о предметной области:

1. Список сущностей предметной области.
2. Список атрибутов сущностей.
3. Описание взаимосвязей между сущностями.

ER-диаграммы удобны тем, что процесс выделения сущностей, атрибутов и связей является итерационным. Разработав первый приближенный вариант диаграмм, мы уточняем их, опрашивая экспертов предметной области. При этом документацией, в которой фиксируются результаты бесед, являются сами ER-диаграммы.

Предположим, что перед нами стоит задача разработать информационную систему по заказу некоторой оптовой торговой фирмы. В первую очередь мы должны изучить предметную область и процессы, происходящие в ней. Для этого необходимо опросить сотрудников фирмы, прочесть документацию, изучить формы заказов, накладных и т. п.

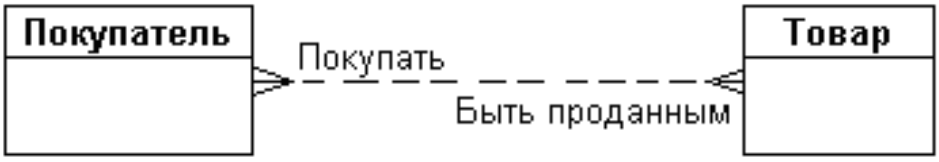
Например, в ходе беседы с менеджером по продажам выяснилось, что он (менеджер) считает, что проектируемая система должна выполнять следующие действия:

- хранить информацию о покупателях;
- печатать накладные на отпущенные товары;
- следить за наличием товаров на складе.

Выделим все существительные в этих предложениях – это будут потенциальные кандидаты на сущности и атрибуты, и проанализируем их (непонятные термины будем выделять знаком вопроса):

- *покупатель* – явный кандидат на сущность;
- *накладная* – явный кандидат на сущность;
- *товар* – явный кандидат на сущность;
- *(?)склад* – а вообще, сколько складов имеет фирма? Если несколько, то это будет кандидатом на новую сущность;
- *(?)наличие товара* – это, скорее всего, атрибут, но атрибут какой сущности?

Сразу возникает очевидная связь между сущностями – «покупатели могут покупать много товаров» и «товары могут продаваться многим покупателям». Первый вариант диаграммы выглядит так:



Задав дополнительные вопросы менеджеру, мы выяснили, что фирма имеет несколько складов. Причем каждый товар может храниться на нескольких складах и быть проданным с любого склада.

Куда поместить сущности «Накладная» и «Склад» и с чем их связать? Выясним, как связаны эти сущности между собой и с сущностями «Покупатель» и «Товар»? Покупатели покупают товары, получая при этом накладные, в которые внесены данные о количестве и цене купленного товара. Каждый покупатель может получить несколько накладных. Каждая накладная обязана выписываться на одного покупателя. Каждая накладная обязана содержать несколько товаров (не бывает пустых накладных). Каждый товар, в свою очередь, может быть продан нескольким покупателям через несколько накладных. Кроме того, каждая накладная должна быть выписана с определенного склада и с любого склада может быть выписано много накладных. Таким образом, после уточнения диаграмма будет выглядеть следующим образом (рис. 5.7):

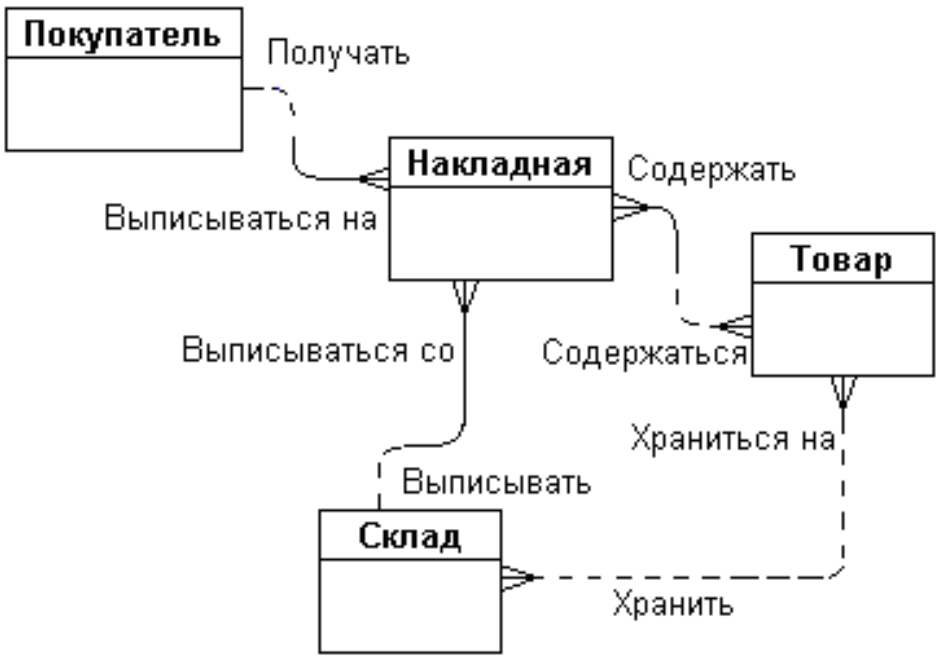


Рис. 5.7. ER-диаграмма

Рассмотрим теперь атрибуты сущностей. Беседуя с сотрудниками фирмы, мы выяснили следующее:

- Каждый покупатель является юридическим лицом и имеет наименование, адрес, банковские реквизиты.
- Каждый товар имеет наименование, цену, а также характеризуется единицами измерения.
- Каждая накладная имеет уникальный номер, дату выписки, список товаров с количествами и ценами, а также общую сумму накладной. Накладная выписывается с определенного склада и на определенного покупателя.
- Каждый склад имеет свое наименование.

Снова выпишем все существительные, которые будут потенциальными атрибутами, и проанализируем их:

- *Юридическое лицо* – термин риторический, мы не работаем с физическими лицами. Не обращаем внимания.
- *Наименование покупателя* – явная характеристика покупателя.
- *Адрес* – явная характеристика покупателя.
- *Банковские реквизиты* – явная характеристика покупателя.
- *Наименование товара* – явная характеристика товара.
- *(?)Цена товара* – похоже, что это характеристика товара. Отличается ли эта характеристика от цены в накладной?
- *Единица измерения* – явная характеристика товара.
- *Номер накладной* – явная уникальная характеристика накладной.
- *Дата накладной* – явная характеристика накладной.
- *(?)Список товаров в накладной* – список не может быть атрибутом. Вероятно, нужно выделить этот список в отдельную сущность.
- *(?)Количество товара в накладной* – это явная характеристика, но характеристика чего? Это характеристика не просто «товара», а «товара в накладной».
- *(?)Цена товара в накладной* – опять же это должна быть не просто характеристика товара, а характеристика товара в накладной. Но цена товара уже встречалась выше – это одно и то же?
- *Сумма накладной* – явная характеристика накладной. Эта характеристика не является независимой. Сумма накладной

равна сумме стоимостей всех товаров, входящих в накладную.

- *Наименование склада* – явная характеристика склада.

В ходе дополнительной беседы с менеджером удается прояснить различные понятия цен. Оказалось, что каждый товар имеет некоторую текущую цену. Эта цена, по которой товар продается в данный момент. Естественно, что эта цена может меняться со временем. Цена одного и того же товара в разных накладных, выписанных в разное время, может быть различной. Таким образом, имеются *две цены* – цена товара в накладной и текущая цена товара.

Сущности «Накладная» и «Товар» связаны друг с другом отношением типа многие-ко-многим. Такая связь, как отмечалось ранее, должна быть преобразована к двум связям типа один-ко-многим.

Для этого требуется дополнительная сущность подчиненного типа как для накладной, так и товара. Этой сущностью и будет сущность «Список товаров в накладной». Связь ее с сущностями «Накладная» и «Товар» характеризуется следующими фразами – «каждая накладная обязана иметь несколько записей из списка товаров в накладной», «каждая запись из списка товаров в накладной обязана включаться ровно в одну накладную», «каждый товар может включаться в несколько записей из списка товаров в накладной», «каждая запись из списка товаров в накладной обязана быть связана ровно с одним товаром». Атрибуты «Количество товара в накладной» и «Цена товара в накладной» являются атрибутами сущности « Список товаров в накладной».

Точно так же поступим со связью, соединяющей сущности «Склад» и «Товар». Введем дополнительную сущность «Товар на складе». Атрибутом этой сущности будет «Количество товара на складе». Таким образом, товар будет числиться на любом складе и количество его на каждом складе будет свое.

Теперь можно внести все это в диаграмму (рис. 5.9).

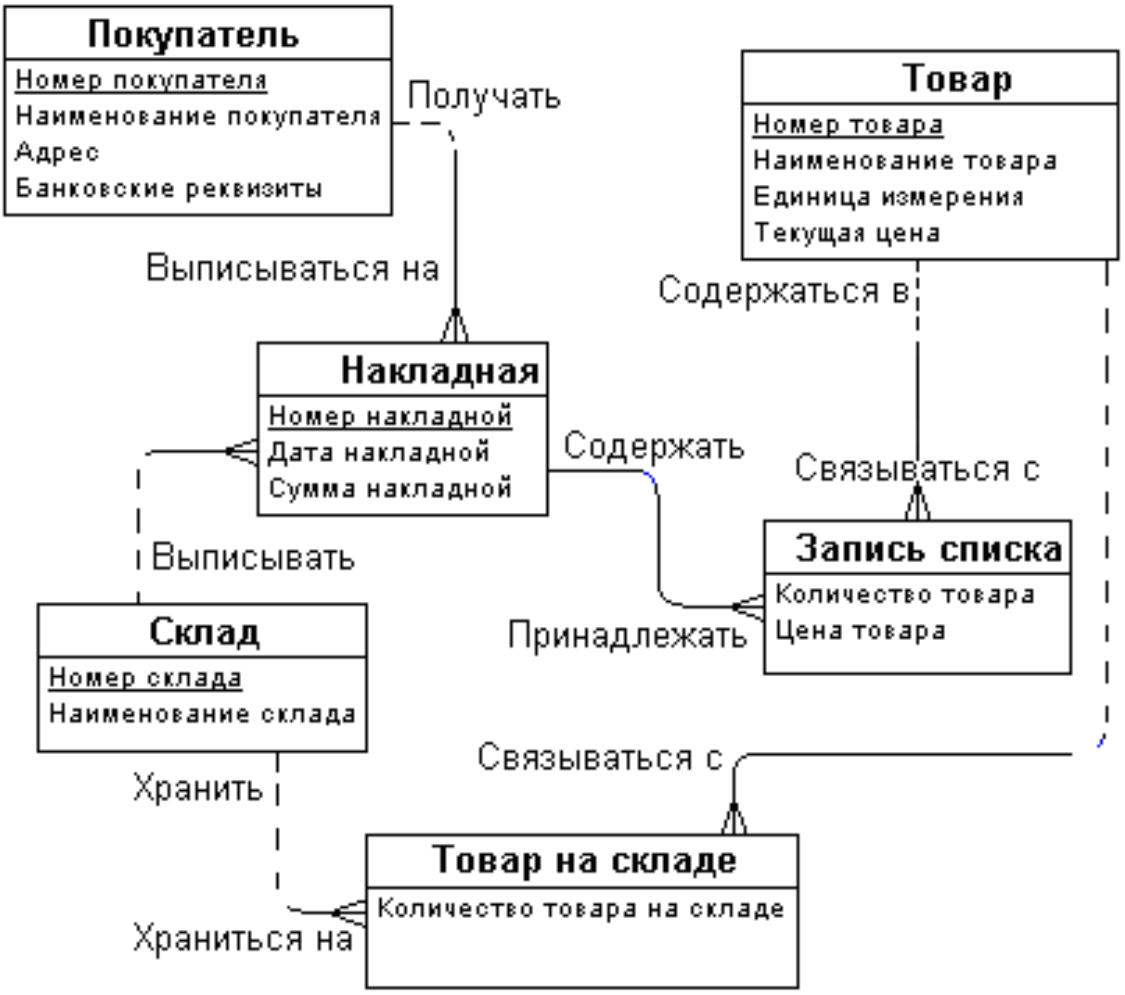


Рис. 5.9. Окончательный вариант ER-диаграммы

5.3. Концептуальные и физические ER-модели

Разработанный выше пример ER-диаграммы является примером ***концептуальной диаграммы***. Это означает, что диаграмма *не учитывает* особенности конкретной СУБД. По данной концептуальной диаграмме можно построить ***физическую диаграмму***, которая уже будут учитываться такие особенности СУБД, как допустимые типы и наименования полей и таблиц, ограничения целостности и т. п.

Правила преобразования ER-диаграммы в физическую модель следующие.

1. Каждой сущности ставится в соответствие отношение реляционной модели данных (таблица). При этом имена сущности и отношения могут быть различными, потому что на имена сущностей обычно не накладываются синтаксические ограничения, кроме уникальности. Имена отношений (таблиц) могут быть

ограничены требованиями конкретной СУБД, чаще всего эти имена являются идентификаторами в некотором базовом языке, они ограничены по длине и не должны содержать пробелов и некоторых специальных символов.

2. Каждый атрибут сущности становится атрибутом соответствующего отношения. Для каждого атрибута задается конкретный допустимый в СУБД тип данных и обязательность или необязательность данного атрибута (т. е. допустимость или недопустимость null-значений).

3. Первичный ключ сущности становится первичным ключом (PRIMARY KEY) соответствующего отношения. Атрибуты первичного ключа автоматически получают свойство обязательности (NOT null).

4. В каждое отношение, соответствующее подчиненной сущности, включается набор атрибутов главной сущности, являющийся первичным ключом главной сущности. В отношении, соответствующем подчиненной сущности, этот набор атрибутов становится внешним ключом (FOREING KEY).

5. При моделировании необязательного типа связи (модальность «может») у атрибутов, соответствующих внешнему ключу, устанавливается свойство допустимости null-значений. При обязательном типе связи атрибуты получают свойство отсутствия неопределенных значений (NOT null).

Физический вариант диаграммы, приведенной на рис. 5.10, может выглядеть, например, следующим образом.

На данной диаграмме каждая сущность представляет собой таблицу базы данных, каждый атрибут становится колонкой соответствующей таблицы. Обращаем внимание на то, что во многих таблицах, например, «CUST_DETAIL» и «PROD_IN_SKLAD», соответствующих сущностям «Запись списка накладной» и «Товар на складе», появились новые атрибуты, которых не было в концептуальной модели, – это ключевые атрибуты главных таблиц, в соответствии с правилом 4 ***мигрировавших*** в подчиненные таблицы для того, чтобы обеспечить связь между таблицами посредством внешних ключей.

Легко заметить, что полученные таблицы сразу находятся в 3НФ.

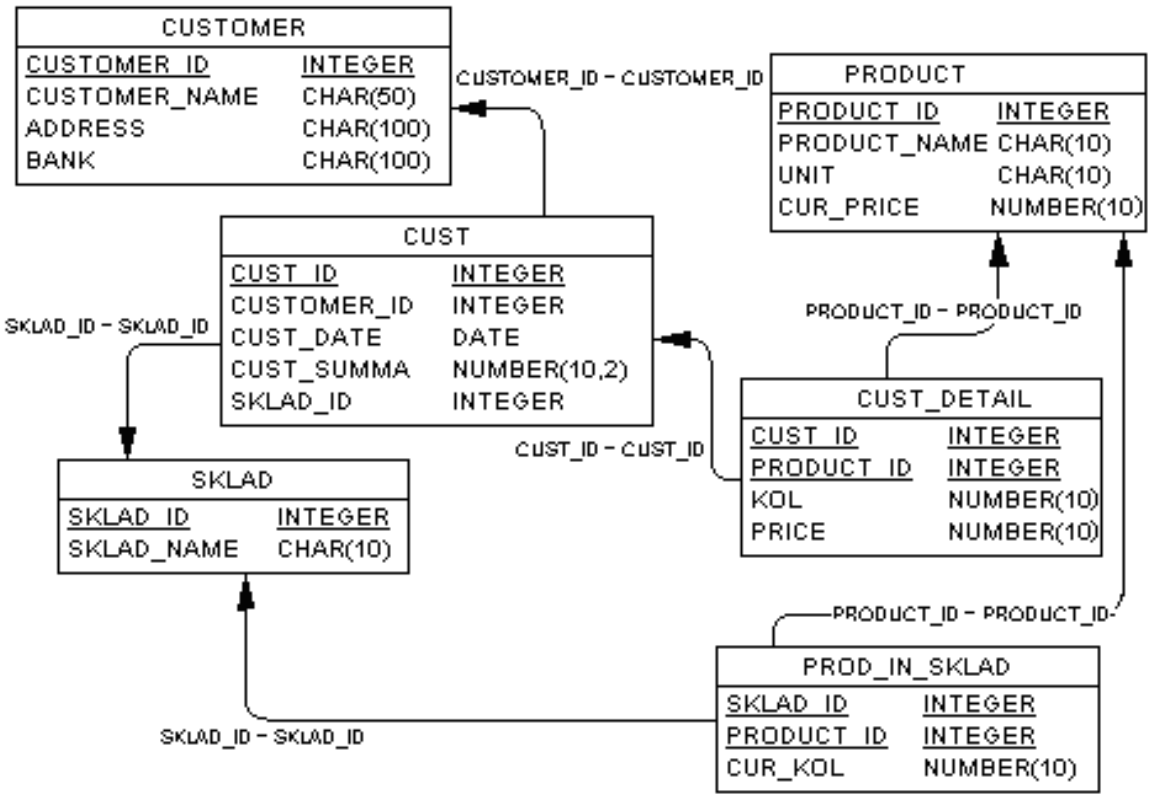


Рис. 5.10. Физический вариант диаграммы

В данной главе, являющейся лишь иллюстрацией к методам ER-моделирования, не рассмотрены более сложные аспекты построения диаграмм, такие как *подтипы*, *роли*, *исключающие связи*, *непереносимые связи*, *идентифицирующие связи* и т. п.

Контрольные вопросы

1. Назовите основные концепции ER-модели и укажите способ их представления на диаграммах.

2. Укажите ограничения, которые могут быть наложены на сущности – участницы некоторой связи.

3. Опишите проблемы, которые могут возникнуть при создании ER-модели.

4. Как можно реализовать связь 1:M в базе данных, состоящей из двух таблиц? Приведите пример.

5. Предположим, что у вас есть ER-модель, представленная на рисунке ниже.

В течение некоторого времени водитель может водить несколько различных грузовиков и каждым грузовиком может управлять несколько водителей. Как можно конвертировать эту модель в ER-модель, на которой будут присутствовать только связи 1:M?

Практическое задание

Рассмотрим проект базы данных банка, содержащей информацию о клиентах (назовем это множество сущностей Customers) и состоянии их счетов (Accounts). Данные о клиенте включают его имя (name), адрес (address), номер телефона (phone) и код полиса пенсионного страхования (kod_polis). Счет описывается атрибутами номера (number), типа (например, «накопительный», «чековый» и т. п.) (type) и остатка (balance). Необходимо также отразить в базе данных факт принадлежности счета определенному клиенту. Начертите ER-диаграмму, соответствующую такой базе данных. Приведите ER-диаграмму к схеме реляционной БД.

6. Транзакции

6.1. Понятие транзакции

Основным предназначением баз данных является предоставление пользователям совокупности накопленных данных для решения прикладных задач. В реляционных базах данных для решения этих задач используются последовательно выполняющиеся команды языка SQL. Если бы в каждый момент времени с базой данных работал только один пользователь, то, закончив выполнение одной задачи, он переходил бы к следующей задаче или передавал возможность работы с базой другому пользователю. Очевидно, однако, что, поскольку любая база данных предназначена для коллективного использования, такой режим работы является крайне неэффективным и СУБД должна предусматривать возможность параллельного выполнения нескольких задач с использованием одной и той же базы данных. При этом, естественно, возникает возможность того, что пользователи будут мешать друг другу, внося в данные изменения, не предсказуемые для других пользователей. Кроме того, даже если с базой данных работает только один пользователь, всегда существует еще один виртуальный «пользователь», называемый *окружающей средой*,

который может вносить непредсказуемые изменения в базу данных за счет отказов оборудования, ошибок в программном обеспечении и т. п. Все это привело к необходимости выделения совокупности команд SQL, обеспечивающих решение одной задачи, в группу, называемую *транзакцией*, которая должна быть выполнена целиком. Если же в ходе выполнения транзакции возникли ошибки или необходимость отказа от выполнения транзакции, то данные, относящиеся к этой транзакции, должны быть возвращены в состояние, которое они имели на момент начала транзакции. Это соответствует тому, что при использовании базы данных пользователя интересует выполнение его задач в целом, а не отдельных команд SQL, которые реализуют выполнение этих задач.

Надо иметь в виду, что в ходе работы транзакции с необходимыми ей данными у других транзакций также могла возникнуть необходимость просматривать или изменять те же самые данные, и СУБД должна корректно разрешать возникающие в такой ситуации проблемы.

Язык SQL содержит несколько команд, обеспечивающих поддержку транзакций, основными из которых являются COMMIT, фиксирующая успешное завершение транзакции, и ROLLBACK, обеспечивающая возврат данных к состоянию на момент начала транзакции. Кроме того, системы управления базами данных содержат средства, позволяющие с той или иной степенью эффективности управлять взаимодействием транзакций с одними и теми же данными.

6.2. Проблемы параллелизма при работе с данными

6.2.1. «Потерянное» обновление

Рассмотрим основные проблемы, возникающие при совместной работе транзакций с данными, условно предполагая, что СУБД никак не вмешивается в процесс выполнения этих транзакций.

Традиционно принято выделять четыре таких проблемы, каждой из которых соответствуют свои методы разрешения.

Первой из них является т. н. «потерянное» обновление (lost update). Для его иллюстрации изобразим последовательность выполнения операторов SQL в двух параллельных задачах в виде следующей таблицы.

Момент времени	Задача 1	Задача 2
1	`UPDATE t1 SET a1 = 5`	
2		`UPDATE t1 SET a1 = 2`
3	`SELECT a1 FROM t1`	

В результате оператор SELECT в первой задаче выдаст значение 2, а не ожидаемое значение 5, т. е. результат выполнения оператора UPDATE в первой задаче будет потерян.

6.2.2. Преждевременное чтение

Другой проблемой является преждевременное (незафиксированное) чтение (dirty read), иллюстрируемое следующей таблицей.

Момент времени	Задача 1	Задача 2
1	`UPDATE t1 SET a1 = 5`	
2	`COMMIT`	
3	`UPDATE t1 SET a1 = 2`	
4		`SELECT a1 FROM t1`
5	`ROLLBACK`	

В этом примере задача 2 прочитала значение a1 до того, как оно было зафиксировано, и в дальнейшем будет использовать ошибочное значение, равное 2, а не восстановленное значение 5.

6.2.3. Неповторяющееся чтение

Продемонстрируем теперь еще одну проблему, заключающуюся в различных результатах выборки данных в разные моменты времени (unrepeatable read).

Момент времени	Задача 1	Задача 2
1	`UPDATE t1 SET a1 = 5`	
2	`COMMIT`	
3		`SELECT a1 FROM t1`
4	`UPDATE t1 SET a1 = 2`	
5	`COMMIT`	
6		`SELECT a1 FROM t1`

В этом примере задача 2 читает только зафиксированные результаты, однако в разные моменты времени один и тот же запрос в задаче 2 дает разные результаты, т. е. с точки зрения этой задачи данные в базе данных являются несогласованными в том смысле, что нельзя определить, какие же из них являются корректными.

6.2.4. Вставка строк-призраков

Разновидностью неповторяющегося чтения является вставка строк-призраков (phantom insert).

Момент времени	Задача 1	Задача 2
1		`SELECT SUM(a1) FROM t1`
2	`INSERT INTO t1 …`	
3		`SELECT SUM(a1) FROM t1`

Здесь также в разные моменты времени один и тот же запрос в задаче 2 дает разные результаты, создавая впечатление с точки зрения этой задачи, что новая строка в таблице t1 появилась «ниоткуда». В отличие от предыдущего примера приведенная проблема может иметь меньший эффект, поскольку если выборка строк ведется по первичному ключу и статистические функции не используются, то задача 2 просто «не увидит» новую строку.

6.3. Свойства транзакций

Сформулируем теперь условия, предъявляемые к транзакциям, которые позволят исключить или скомпенсировать описанные в предыдущем пункте нежелательные эффекты. Обычно их обозначают аббревиатурой ACID – Atomicity, Consistency, Isolation, Durability (неделимость, согласованность, изолированность, устойчивость). Означают они следующее.

- Atomicity – действие транзакции по принципу «все или ничего»: либо база данных переходит в новое состояние, соответствующее выполненным в ходе транзакции модификациям базы данных, либо, если в ходе выполнения возникли критические ошибки или пользователь решил прервать выполнение транзакции, – возврат базы данных к состоянию на момент начала транзакции.
- Consistency – перевод из одного целостного (согласованного) состояния в другое. При этом в ходе выполнения отдельных команд транзакции согласованность данных может нарушаться (например, хранимое значение суммы каких-либо данных может не соответствовать реальной сумме этих данных), однако по ее окончании все данные, относящиеся к этой транзакции, должны соответствовать друг другу. Внутренние структуры данных, поддерживающие согласованность дан-

ных, например индексы или связи «родитель-потомок», также должны быть правильными в конце транзакции.

- Isolation (изолированность) – недоступность промежуточных результатов другим пользователям. У пользователя должна создаваться иллюзия, что только он работает с базой данных. Единственным эффектом от присутствия других пользователей может быть лишь замедление работы системы (иногда весьма значительное). Это свойство называют также упорядочиваемостью, поскольку при параллельном выполнении транзакций оно дает тот же самый результат, что и при их последовательном выполнении в некотором порядке.
- Durability (устойчивость) – сохранение зафиксированных данных. Это означает, что в конце транзакции ее результаты должны быть переписаны из буферов оперативной памяти на диски, чтобы дальнейшие события, происходящие в процессе функционирования базы данных, не могли изменить результаты этой транзакции. Разумеется, данные, с которыми оперировала транзакция, впоследствии могут быть изменены, но при откате на момент окончания транзакции они должны быть теми же, что получились в результате ее фиксации.

6.4. Реализация транзакций в СУБД

6.4.1. Журнал транзакций

Основным средством, обеспечивающим атомарность транзакций, является ведение журнала транзакций. СУБД записывает в журнал все SQL-команды, выполняемые в ходе транзакции, а также состояние данных до и после выполнения команды. В этом случае для отката на начало транзакции следует «выполнить» эти команды в обратном порядке, последовательно восстанавливая предыдущие состояния данных. Очевидно, что ведение журнала требует значительных накладных расходов и поэтому в реальных СУБД имеются средства для управления журналом транзакций, позволяющие выбрать наиболее эффективный вариант, вплоть до полного отключения журнала.

6.4.2. Управление транзакциями

Для формирования транзакций СУБД обычно поддерживает несколько режимов. Простейшим из них является автоматический режим, в котором каждая SQL-команда рассматривается как отдельная транзакция. Если команда выполнена успешно, СУБД выполняет действия, соответствующие команде COMMIT, в противном случае – команде ROLLBACK. При этом сами команды COMMIT и ROLLBACK не пишутся.

В случае, когда в транзакцию требуется объединить несколько SQL-команд, используется явный или неявный режим указания транзакций.

В неявном режиме запуск приложения, работающего с базой данных, открывает первую транзакцию, а последующие команды COMMIT или ROLLBACK завершают текущую транзакцию и открывают следующую. Успешное завершение приложения вызывает действия, соответствующие команде COMMIT, а аварийное – команде ROLLBACK.

В явном режиме транзакция открывается командой BEGIN TRANSACTION, а завершается теми же командами COMMIT или ROLLBACK, после чего система обычно переходит в автоматический режим.

Возвращаясь к свойствам транзакции, можно сказать, что команда ROLLBACK обеспечивает атомарность транзакции, а команда COMMIT – ее устойчивость, поскольку по этой команде освобождаются используемые транзакцией ресурсы, а буфера данных сбрасываются на диск и тем самым происходит фиксация результатов транзакции.

6.4.3. Блокировки

Основным способом реализации изолированности транзакций является блокировка используемых транзакцией ресурсов, т. е. установка режима приостановки доступа других транзакций к этим ресурсам. Основными характеристиками блокировки являются:

- период времени, на который устанавливается блокировка;
- уровень (объем) блокируемых данных;
- вид блокировки.

Период блокировки обычно устанавливается от запроса требуемого ресурса и до конца транзакции. Это, в частности, означает, что транзакции следует делать максимально короткими. В то же время во многих случаях возможен и такой режим, когда команды блокирования и разблокирования выполняются явным образом.

Уровень блокируемых данных определяется как решаемыми задачами, так и соотношением между выигрышем от повышения параллельного выполнения транзакций и требуемыми для этого накладными расходами. Самым грубым вариантом является блокировка всей базы данных, применяемая, например, при ее реструктуризации и блокирующая выполнение всех остальных транзакций, а потому редко применяемая. На логическом уровне можно выделить блокировку таблиц, отдельных строк и диапазонов строк (определяемых, например, командами SELECT … WHERE …). Однако, поскольку накладные расходы на реализацию блокировки строк достаточно велики, чаще блокируются *страницы* (блоки данных на диске), на которых располагаются эти строки.

К видам блокировки относят разделяемую и исключительную блокировки. Разделяемая блокировка (блокировка записи) означает, что блокируются только команды модификации данных (INSERT, DELETE, UPDATE), а команды чтения (SELECT) разрешены. Исключительная блокировка (блокировка чтения) запрещает все команды работы с блокируемыми данными.

6.5. Уровни изоляции

Язык SQL предусматривает настройку взаимодействия транзакций, называемую уровнями изоляции. Мы уже отмечали, что свойство изолированности транзакции предусматривает такой режим работы, при котором результаты параллельного выполнения нескольких транзакций оказываются такими же, как и при последовательном их выполнении, причем в любом порядке. Этого можно добиться, если каждая транзакция будет от начала до конца транзакции устанавливать исключительную блокировку на все требуемые ей ресурсы. Понятно, что использование ресурсов будет при этом достаточно неэффективным и выигрыш от параллельной работы транзакций будет невелик. Поэтому в случае, когда некото-

рые из ситуаций, описанных в п. 6.2, не являются значимыми, можно уменьшить степень изолированности транзакций друг от друга, уменьшив тем самым объем блокируемых ресурсов и повысив эффект от параллельного выполнения транзакций.

Язык SQL предусматривает несколько уровней изоляции транзакций, ослабляющих требования к их взаимодействию. Понижение требований относится только к операциям чтения, так что выполнение модификации данных блокирует их в исключительном режиме.

SERIALIZABLE – наиболее высокий уровень изоляции, обеспечивающий те же результаты, что и при последовательном исполнении транзакций. В исключительном режиме блокируются все строки и диапазоны строк, которые читались или модифицировались в текущей транзакции. Кроме того, блокируется доступ этой транзакции к строкам и диапазонам строк, которые читались или модифицировались в других транзакциях. Этот уровень гарантирует отсутствие всех нежелательных эффектов от взаимодействия транзакций, описанных в п. 6.2. По существу, только этот уровень изоляции обеспечивает классические свойства транзакций (ACID).

REPEATABLE READ – несколько меньший по степени изоляции уровень. На этом уровне блокируются (в исключительном режиме) строки (не диапазоны строк), использовавшиеся в запросах текущей транзакции, однако допускается вставка строк другими транзакциями, что может привести к появлению строк-призраков (п. 6.2.4). Это, однако, допустимо, если в текущей транзакции не используются запросы со статистическими функциями и подобные им. Также блокируется доступ текущей транзакции к строкам, которые читались или модифицировались другими транзакциями.

Уровень READ COMMITTED (чтение с фиксацией) блокирует (в исключительном режиме) строки, которые вставлялись, удалялись или модифицировались. Этот режим допускает неповторяющееся чтение и вставку «призраков», но блокирует преждевременное («грязное») чтение и потерянное обновление.

Уровень READ UNCOMMITTED блокирует строки, которые вставлялись, удалялись или модифицировались, в неисключительном режиме и обеспечивает лишь отсутствие потерянных обновлений.

Команда, устанавливающая уровень изоляции, обычно выполняется в самом начале транзакции и имеет вид

```
SET TRANSACTION ISOLATION LEVEL <уровень изоляции>
```

Выбор наиболее подходящего данной транзакции уровня изоляции может значительно уменьшить периоды ожидания транзакций, вызванные блокировками.

6.6. Взаимоблокировки

Взаимоблокировка возникает, когда две или более задач постоянно блокируют друг друга в ситуации, когда у каждой задачи заблокирован ресурс, который пытаются заблокировать другие задачи. Например:

Момент времени	Задача 1	Задача 2
1	`SELECT * FROM t1`	
2		`SELECT * FROM t2`
3	`UPDATE t2 SET a2 = 5`	
4		`UPDATE t1 SET a1 = 2`
5	`COMMIT`	
6		`COMMIT`

Здесь вначале задача 1 блокирует таблицу t1, а задача 2 блокирует таблицу t2. После этого задача 1 пытается изменить таблицу t2, но не может этого сделать из-за блокировки t2 задачей 2 и поэтому ожидает завершения задачи 2. В свою очередь, задача 2 по тем же причинам ожидает завершения задачи 1. Поэтому обе задачи находятся в состоянии блокировки и будут находиться в состоянии ожидания до тех пор, пока взаимоблокировка не будет снята внешним процессом, например принудительным завершением какой-либо задачи.

Отметим две особенности приведенного примера. Во-первых, предполагается, что обе транзакции выполняются в явном или неявном, а не в автоматическом режиме, иначе каждая команда фиксируется и не создает блокировки. Во-вторых, для возникновения блокировок необходимо, чтобы уровень изоляции транзакций был не ниже REPEATABLE READ. На более низких уровнях изоляции блокировка отсутствует, и обе транзакции завершатся успешно.

Для отслеживания и устранения взаимоблокировок в состав СУБД обычно входит специальная системная программа (мони-

тор блокировок), которая периодически (например, каждые 5 секунд) проверяет блокировки, установленные всеми транзакциями, и если находит взаимоблокировку, то принудительно аварийно завершает одну из соответствующих транзакций. В простейшем же варианте чаще всего можно установить режим, в котором транзакция завершается, если время ожидания освобождения требуемых ресурсов превышает заданную величину (тайм-аут).

Хотя полностью избежать взаимоблокировок нельзя, для их минимизации можно использовать следующие приемы:

- выполнять доступ к объектам в одинаковом порядке, в частности, в приведенном примере задача 2 должна сначала обработать таблицу t1, а уже затем – t2;
- минимизировать размер транзакций;
- не включать взаимодействие с пользователем в транзакции;
- использовать наиболее низкий уровень изоляции, приемлемый для данной задачи.

6.7. Технология версий

Описанную выше систему блокировок называют пессимистическим управлением параллелизмом, поскольку она предполагает, что параллельные транзакции будут обязательно мешать друг другу. В то же время в случаях, когда различные транзакции работают в основном с различными блоками данных, такой подход ведет к неоправданно большим накладным расходам на поддержание системы блокировок. В связи с этим получил распространение альтернативный подход, называемый технологией версий, или оптимистическим управлением параллелизмом. В этом случае пользователи не блокируют данные на период чтения. Когда же пользователь обновляет, например, строку данных, система создает две копии (версии) этой строки, одна из которых содержит старые данные, другая – новые. В зависимости от уровня изоляции система выдает другим транзакциям ту или иную версию. После фиксации транзакции остается лишь одна версия данных. При попытке же обновления незафиксированных данных другой транзакцией возникает ошибка, и последняя транзакция должна быть повторена заново.

Оптимистическое управление в основном применяется в системах с небольшим количеством конфликтов данных, где затраты

на периодический откат транзакции меньше затрат на блокировку данных при считывании. В то же время пессимистическое управление (блокировки) используется в средах с большим количеством конфликтов данных, где затраты на защиту данных с помощью блокировок меньше затрат на откат транзакций в случае конфликтов параллелизма.

6.8. Распределенные транзакции

Если транзакция использует данные, размещенные на нескольких серверах, то она называется распределенной транзакцией. Управление распределенными транзакциями имеет свои особенности, связанные с тем, что во время выполнения транзакции некоторые сервера могут оказаться недоступными.

В приложении управление распределенной транзакцией аналогично управлению локальной транзакцией. В конце транзакции приложение запрашивает ее фиксацию или откат. Однако в случае распределенной транзакции управление фиксацией должно строиться иначе, поскольку в случае сбоя сети может оказаться так, что одни серверы будут фиксировать транзакцию, тогда как другие будут выполнять ее откат. Для решения этой проблемы используется двухфазная фиксация, состоящая из фазы подготовки и фазы фиксации.

В фазе подготовки сервер, получивший запрос на фиксацию (ведущий сервер), отправляет команду подготовки всем серверам, занятым в транзакции. Каждый из них выполняет действия по завершению транзакции, а все буферы данных и служебная информация переписываются из оперативной памяти на диск. По мере того как каждый сервер завершает фазу подготовки, он возвращает запросившему фиксацию серверу значение успешного или неуспешного завершения подготовки.

Если запросивший фиксацию сервер получает значения успешного завершения подготовки от всех диспетчеров ресурсов, то он начинает фазу фиксации и отправляет команду фиксации каждому участвующему в транзакции серверу, после чего они могут завершить фиксацию. Если все серверы сообщают об успешной фиксации, то ведущий сервер отправляет уведомление приложению об успешной фиксации. Если же какой-либо сервер сообщил о неуспешном завершении подготовки, то ведущий

сервер отправляет команду отката всем остальным серверам и сообщает приложению о сбое фиксации.

6.9. Заключительные замечания

Настройка системы транзакций является одним из важнейших элементов создания высокоэффективной многопользовательской системы. Возможности различных СУБД различаются в этой области весьма значительно, и поэтому в данной главе приведены только самые общие сведения о транзакциях. Кроме того, эффективность системы зависит не только от администраторов БД, задающих эти настройки, но и от разработчиков, которые должны обеспечить высокую степень параллелизма и корректность транзакций, гарантирующих согласованность данных по завершении каждой транзакции. И конечно же, для создания качественной системы необходимо глубоко изучить документацию по применяемой СУБД.

Контрольные вопросы

1. Опишите основные проблемы, возникающие при многопользовательской работе с базой данных.
2. Назовите основные требования, предъявляемые к транзакциям.
3. Каковы основные команды SQL для управления транзакциями и режимы их использования?
4. Какие средства используются для реализации системы управления транзакциями?
5. Что такое уровни изоляции транзакций? Какому уровню соответствует транзакция, удовлетворяющая ACID-условиям?
6. В каких случаях целесообразно использовать оптимистическое управление транзакциями?

Практическое задание

Изучите систему управления транзакциями в какой-либо конкретной СУБД: используемые команды SQL, реализованные уровни изоляции, блокируемые объекты, управление журнализацией, наличие/отсутствие оптимистического управления блокировкой.

7. Технологии клиент-сервер

7.1. Сервер базы данных

Первоначально СУБД имели централизованную архитектуру, определяемую архитектурой вычислительной техники: центральным компьютером (большая ЭВМ или мини-компьютер) и подключенными к нему терминалами, выступавшими в качестве рабочих мест пользователей (рис. 7.1.). При этом пользовательские терминалы не имели собственных ресурсов, то есть процессоров и памяти, которые могли бы использоваться для хранения и обработки данных. В данной архитектуре сама СУБД и прикладная программа, работающая с базой данных, функционировали на центральном компьютере. Там же располагались базы данных. Данная архитектура предъявляла жесткие требования к производительности центрального компьютера. Особенности СУБД того поколения напрямую связаны с архитектурой систем больших ЭВМ и мини-компьютеров и адекватно отражают все их преимущества и недостатки.

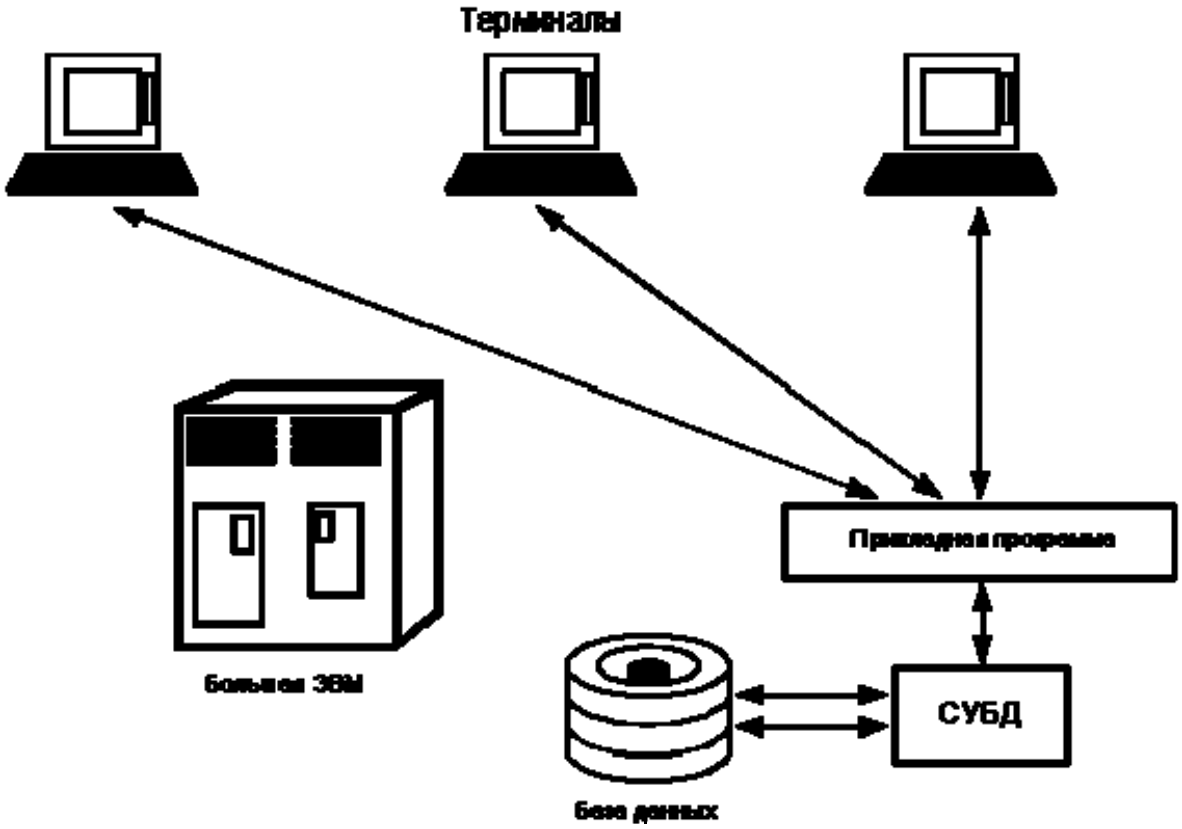


Рис. 7.1. Системы с централизованной архитектурой

Таким образом, ранее пользовательская программа не разделялась на части, она выполнялась некоторым монолитным блоком. Общая тенденция развития вычислительной техники от отдельных mainframe-систем к вычислительным системам, объединяющим компьютеры, привела к необходимости решения задачи более рационального использования ресурсов. Действительно, все компьютеры в сети обладают собственными ресурсами, и разумно так распределить нагрузку на них, чтобы максимальным образом использовать их ресурсы. Для воплощения идеи была разработана модель разбиения единого монолитного приложения на отдельные части и определены принципы взаимосвязи между этими частями.

Применительно к технологии баз данных функции стандартного интерактивного приложения разделяются на 5 групп, имеющих различную природу:

- функции ввода и отображения данных (Presentation Logic);
- прикладные функции, характерные для данной предметной области (Business Logic);
- функции хранения и управления информационными ресурсами внутри приложения (Data manipulation Logic);
- фундаментальные функции управления информационными ресурсами (Database Manager System);
- служебные функции.

Первый вид функций (Presentation Logic) как часть приложения определяется тем, что пользователь видит на своем экране, когда работает приложение. Сюда относятся все интерфейсные экранные формы, которые пользователь видит или заполняет в ходе работы приложения, к этой же части относится все то, что выводится пользователю на экран как результаты решения некоторых промежуточных задач либо как справочная информация. Для организации презентационной логики используется знакоориентированный или графический пользовательский интерфейс.

Второй вид (Business Logic) – это часть кода приложения, которая определяет собственно алгоритмы решения конкретных задач приложения (например, для банковской системы – открытие счета, перевод денег с одного счета на другой и т. д.). Обычно этот код пишется с использованием различных языков программирования.

Третий вид (Data manipulation Logic) – это часть кода приложения, которая связана с обработкой данных внутри приложения.

Данными управляет собственно СУБД. Для обеспечения доступа к данным используются язык запросов и средства манипулирования данными стандартного языка SQL. Обычно операторы языка SQL встраиваются в языки 3 или 4-го поколения (3GL, 4GL), которые используются для написания кода приложения.

Четвертый вид функций (Database Manager System) — это собственно СУБД, которая обеспечивает хранение и управление базами данных. В идеале функции СУБД должны быть скрыты от бизнес-логики приложения, однако с точки зрения архитектуры приложения они выделяются в отдельную часть приложения.

Служебные функции играют роль связок между функциями первых четырех групп.

В соответствии с этим в любом приложении выделяются следующие логические компоненты: компонент представления, реализующий функции первой группы; прикладной компонент, поддерживающий функции второй группы; компонент доступа к информационным ресурсам, поддерживающий функции третьей группы, а также вводятся и уточняются соглашения о способах их взаимодействия (протокол взаимодействия).

7.2. Технология и модели «клиент-сервер»

В настоящее время наиболее распространенной является технология «клиент-сервер». Для многопользовательских СУБД архитектура «клиент-сервер» стала фактическим стандартом. При этом термины «сервер» и «клиент» могут быть связаны как с вычислительной техникой, так и с программным обеспечением.

«Клиент-сервер» – это модель взаимодействия компьютеров в сети. Как правило, компьютеры не являются равноправными. Каждый из них имеет свое, отличное от других, назначение, играет свою роль. Некоторые компьютеры в сети владеют и распоряжаются информационно-вычислительными ресурсами, такими как процессоры, файловая система, почтовая служба, служба печати, база данных. Другие же компьютеры имеют возможность обращаться к этим службам, пользуясь услугами первых. Компьютер, управляющий тем или иным ресурсом, принято называть сервером этого ресурса, а компьютер, желающий им воспользоваться, – клиентом. Конкретный сервер определяется видом ресурса, которым он владеет. Так, если ресурсом являются базы

данных, то речь идет о сервере баз данных, назначение которого – обслуживать запросы клиентов, связанные с обработкой данных; если ресурс – это файловая система, то говорят о файловом сервере, или файл-сервере, и т. д.

В сети один и тот же компьютер может выполнять роль как клиента, так и сервера. Например, в информационной системе, включающей персональные компьютеры, большую ЭВМ и мини-компьютер под управлением UNIX, последний может выступать как в качестве сервера базы данных, обслуживая запросы от клиентов – персональных компьютеров, так и в качестве клиента, направляя запросы большой ЭВМ.

Этот же принцип распространяется и на взаимодействие программ. Если одна из них выполняет некоторые функции, предоставляя другим соответствующий набор услуг, то такая программа выступает в качестве сервера. Программы, которые пользуются этими услугами, принято называть клиентами. Так, ядро реляционной SQL-ориентированной СУБД часто называют сервером базы данных, или SQL-сервером, а программу, обращающуюся к нему за услугами по обработке данных – SQL-клиентом.

Если предполагается, что проектируемая информационная система (ИС) будет иметь технологию «клиент-сервер», то это означает, что прикладные программы, реализованные в ее рамках, будут иметь распределенный характер. Иными словами, часть функций прикладной программы (или, проще, приложения) будет реализована в программе-клиенте, другая – в программе-сервере, причем для их взаимодействия будет определен некоторый протокол.

Если принять, что в любом приложении информационной системы, работающей с СУБД, выделяются указанные выше три компоненты, то можно указать четыре фактора, определяющие различия в реализациях технологии «клиент-сервер»:

во-первых, тем, в какие виды программного обеспечения интегрированы каждый из этих компонентов;

во-вторых, тем, какие механизмы программного обеспечения используются для реализации функций всех трех групп;

в-третьих, как логические компоненты распределяются между компьютерами в сети;

в-четвертых, какие механизмы используются для связи компонентов между собой.

Выделяются четыре подхода, реализованные в моделях:

модель файлового сервера (File Server – FS);

модель доступа к удаленным данным (Remote Data Access – RDA);

модель севера базы данных (DataBase Server – DBS);

модель сервера приложений (Application Server – AS).

Модель файлового сервера

FS-модель является базовой для локальных сетей персональных компьютеров. Не так давно она была исключительно популярной среди отечественных разработчиков, использовавших такие системы, как FoxPRO, Clipper, Clarion, Paradox и т. д. Суть модели проста и всем известна. Один из компьютеров в сети считается файловым сервером и предоставляет услуги по обработке файлов другим компьютерам. Файловый сервер работает под управлением сетевой операционной системы и играет роль компонента доступа к информационным ресурсам (то есть к файлам). На других компьютерах в сети функционирует приложение, в кодах которого совмещены компонент представления и прикладной компонент (рис. 7.2). Протокол обмена представляет собой набор низкоуровневых вызовов, обеспечивающих приложению доступ к файловой системе на файл-сервере.

FS-модель послужила фундаментом для расширения возможностей персональных СУБД в направлении поддержки многопользовательского режима. В таких системах на нескольких персональных компьютерах выполняется как прикладная программа, так и копия СУБД, а базы данных содержатся в разделяемых файлах, которые находятся на файловом сервере. Когда прикладная программа обращается к базе данных, СУБД направляет запрос на файловый сервер. В этом запросе указаны файлы, где находятся запрашиваемые данные. В ответ на запрос файловый сервер направляет по сети требуемый блок данных. СУБД, получив его, выполняет над данными действия, которые были декларированы в прикладной программе.

Достоинства этой модели в том, что мы уже имеем разделение монопольного приложения на два взаимодействующих процесса. При этом сервер (серверный процесс) может обслуживать множество клиентов, которые обращаются к нему с запросами.

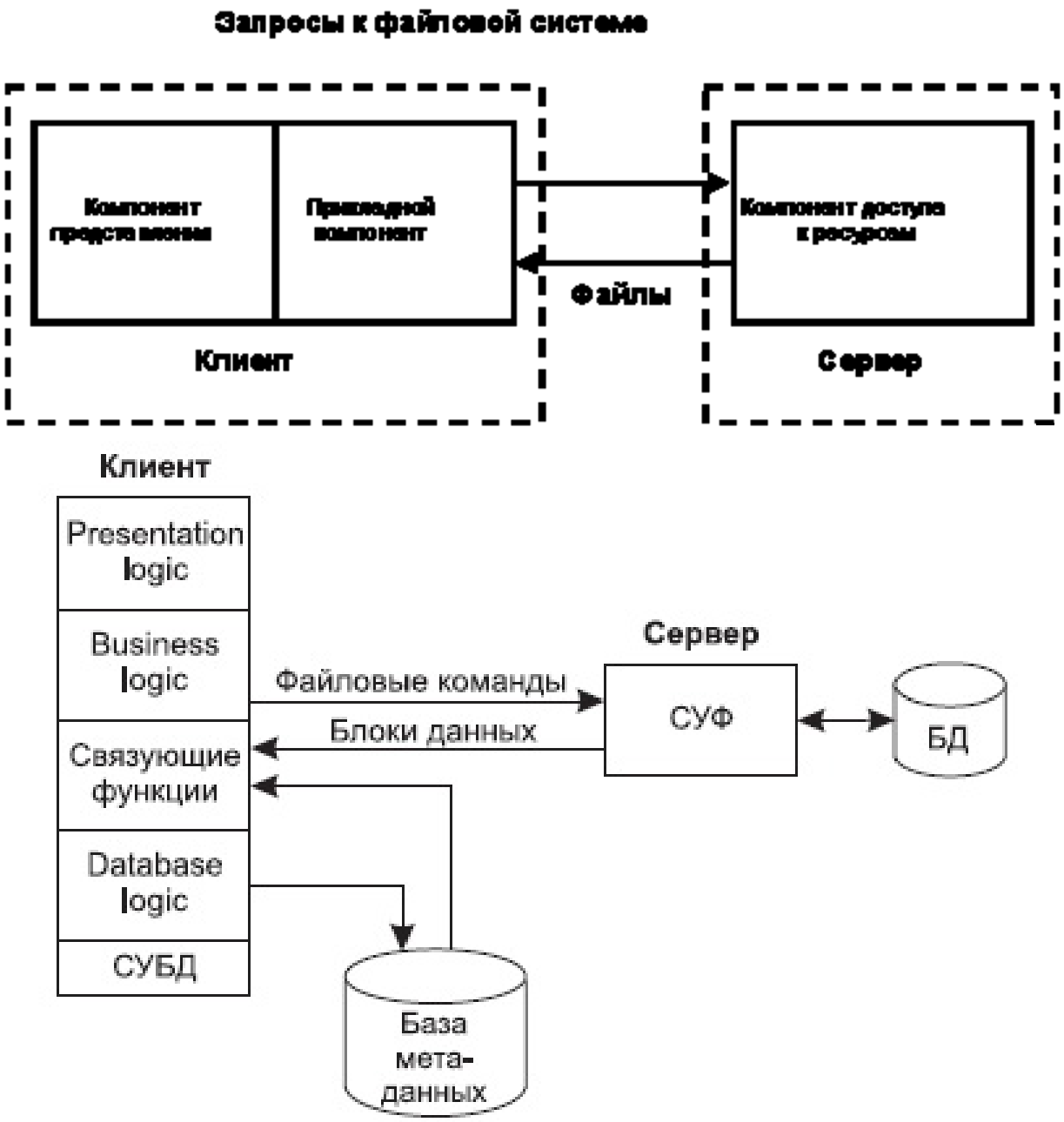


Рис. 7.2. Модель файлового сервера (два варианта)

К технологическим недостаткам модели относят высокий сетевой трафик (передача множества файлов, необходимых приложению), узкий спектр операций манипуляции с данными («данные – это файлы»), отсутствие адекватных средств безопасности доступа к данным (защита только на уровне файловой системы) и т. д. Собственно, перечисленное не есть недостатки, но следствие внутренне присущих FS-модели ограничений, определяемых ее характером. Недоразумения возникают, когда FS-модель используют не по назначению, например, пытаются интерпретировать как модель сервера базы данных. Место FS-модели в

иерархии моделей «клиент-сервер» – это место модели файлового сервера, и ничего более. Именно поэтому обречены на провал попытки создания на основе FS-модели крупных корпоративных систем – попытки, которые предпринимались в недавнем прошлом и нередко предпринимаются сейчас.

Модель доступа к удаленным данным

Более технологичная RDA-модель существенно отличается от FS-модели характером компонента доступа к информационным ресурсам. Это, как правило, SQL-сервер. В RDA-модели коды компонента представления и прикладного компонента совмещены и выполняются на компьютере-клиенте. Последний поддерживает как функции ввода и отображения данных, так и чисто прикладные функции. Доступ к информационным ресурсам обеспечивается либо операторами специального языка (языка SQL, например, если речь идет о базах данных), либо вызовами функций специальной библиотеки (если имеется соответствующий интерфейс прикладного программирования – API).

Клиент направляет запросы к информационным ресурсам (например, к базам данных) по сети удаленному компьютеру. На нем функционирует ядро СУБД, которое обрабатывает запросы, выполняя предписанные в них действия, и возвращает клиенту результат, оформленный как блок данных (рис. 7.3). При этом инициатором манипуляций с данными выступают программы, выполняющиеся на компьютерах-клиентах, в то время как ядру СУБД отводится пассивная роль – обслуживание запросов и обработка данных. Такое распределение обязанностей между клиентами и сервером базы данных не догма – сервер БД может играть более активную роль, чем та, которая предписана ему традиционной парадигмой.

RDA-модель избавляет от недостатков, присущих системам как с централизованной архитектурой, так и с файловым сервером.

При сравнении с централизованной архитектурой можно отметить, что перенос компонента представления и прикладного компонента на компьютеры-клиенты существенно разгружает сервер БД, сводя к минимуму общее число процессов операционной системы. Сервер БД освобождается от несвойственных ему функций; процессор или процессоры сервера целиком загружа-

ются операциями обработки данных, запросов и транзакций. Это становится возможным благодаря отказу от терминалов и оснащению рабочих мест компьютерами, которые обладают собственными локальными вычислительными ресурсами, полностью используемыми программами переднего плана. Впрочем, указанное преимущество можно частично отнести и к модели файлового сервера, хотя, как правило, компьютер-сервер зачастую в данной модели использовался и в качестве клиента.

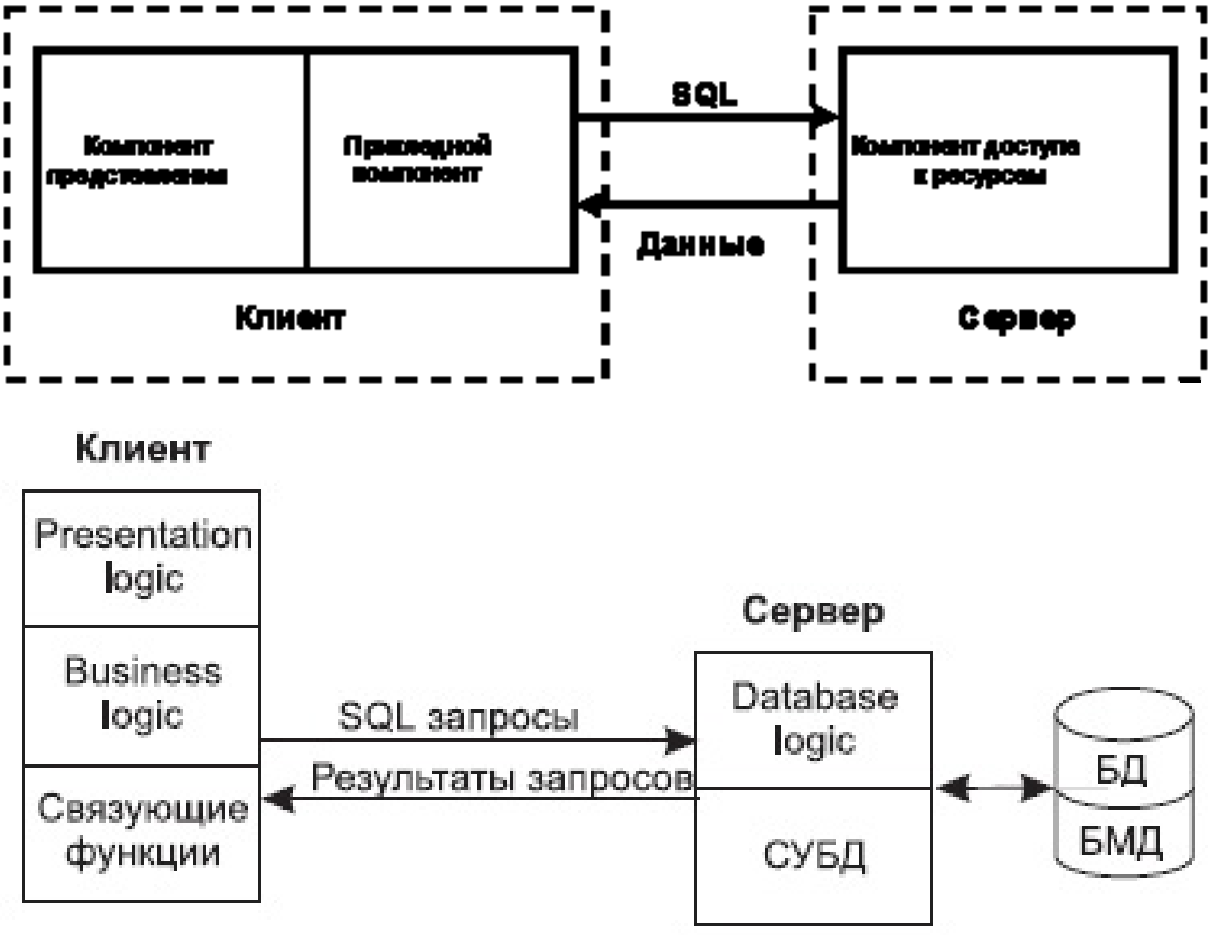


Рис. 7.3. Модель доступа к удаленным данным (два варианта)

При сравнении с моделью файлового сервера следует отметить резкое уменьшение загрузки сети. От клиента к серверу по сети передаются не запросы на ввод-вывод (как в системах с файловым сервером), а запросы на языке SQL, их объем существенно меньше. В ответ на запросы клиент получает только данные, релевантные запросу, а не блоки файлов, как в FS-модели.

Основное достоинство RDA-модели – унификация интерфейса «клиент-сервер» в виде языка SQL. Действительно, взаимо-

действие прикладного компонента с ядром СУБД невозможно без стандартизованного средства общения. Запросы, направляемые программой ядру, должны быть понятны обоим. Для этого их следует сформулировать на специальном языке. Но в СУБД уже существует язык SQL, о котором говорилось ранее. Поэтому целесообразно использовать его в качестве не только средства доступа к данным, но и стандарта общения клиента и сервера.

Такое общение можно сравнить с беседой нескольких человек, когда один отвечает на вопросы остальных (вопросы задаются одновременно). Причем делает это он так быстро, что время ожидания ответа приближается к нулю. Высокая скорость общения достигается прежде всего благодаря четкой формулировке вопроса, когда спрашивающему и отвечающему не нужно дополнительных консультаций по сути вопроса. Беседующие обмениваются несколькими короткими однозначными фразами, им ничего не нужно уточнять.

К сожалению, RDA-модель не лишена ряда недостатков. Во-первых, взаимодействие клиента и сервера посредством SQL-запросов при интенсивной работе клиентских приложений может существенно загрузить сеть. Во-вторых, удовлетворительное администрирование приложений в RDA-модели практически невозможно из-за совмещения в одной программе различных по своей природе функций (функции представления и прикладные).

Модель сервера базы данных

Наряду с RDA-моделью все большую популярность приобретает перспективная DBS-модель (рис. 7.4). Последняя реализована в некоторых реляционных СУБД (Informix, Ingres, Sybase, Oracle). Основу данной модели составляют: механизм хранимых процедур как средство программирования SQL-сервера, механизм триггеров как механизм отслеживания текущего состояния информационного хранилища и механизм ограничений на пользовательские типы данных, который иногда называется механизмом поддержки доменной структуры. Процедуры и триггеры хранятся в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, где функционирует SQL-сервер. Язык, на котором разрабатываются хранимые процедуры, представляет собой процедурное расширение языка запросов SQL и уникален для каждой конкретной

СУБД. Триггеры позволяют организовать постоянный контроль за состоянием БД с целью отслеживания всех вносимых в базу изменений и адекватной реакции на них. Механизм поддержки доменной структуры позволяет пополнять список стандартно допустимых типов данных, что может быть использовано для учета семантической составляющей данных в реальных предметных областях, например координаты объектов или единицы различных метрик. Сервер базы данных часто называют активным сервером, потому что не только клиент, но и сам сервер может быть инициатором обработки данных в БД. Более подробно он будет рассмотрен позже.

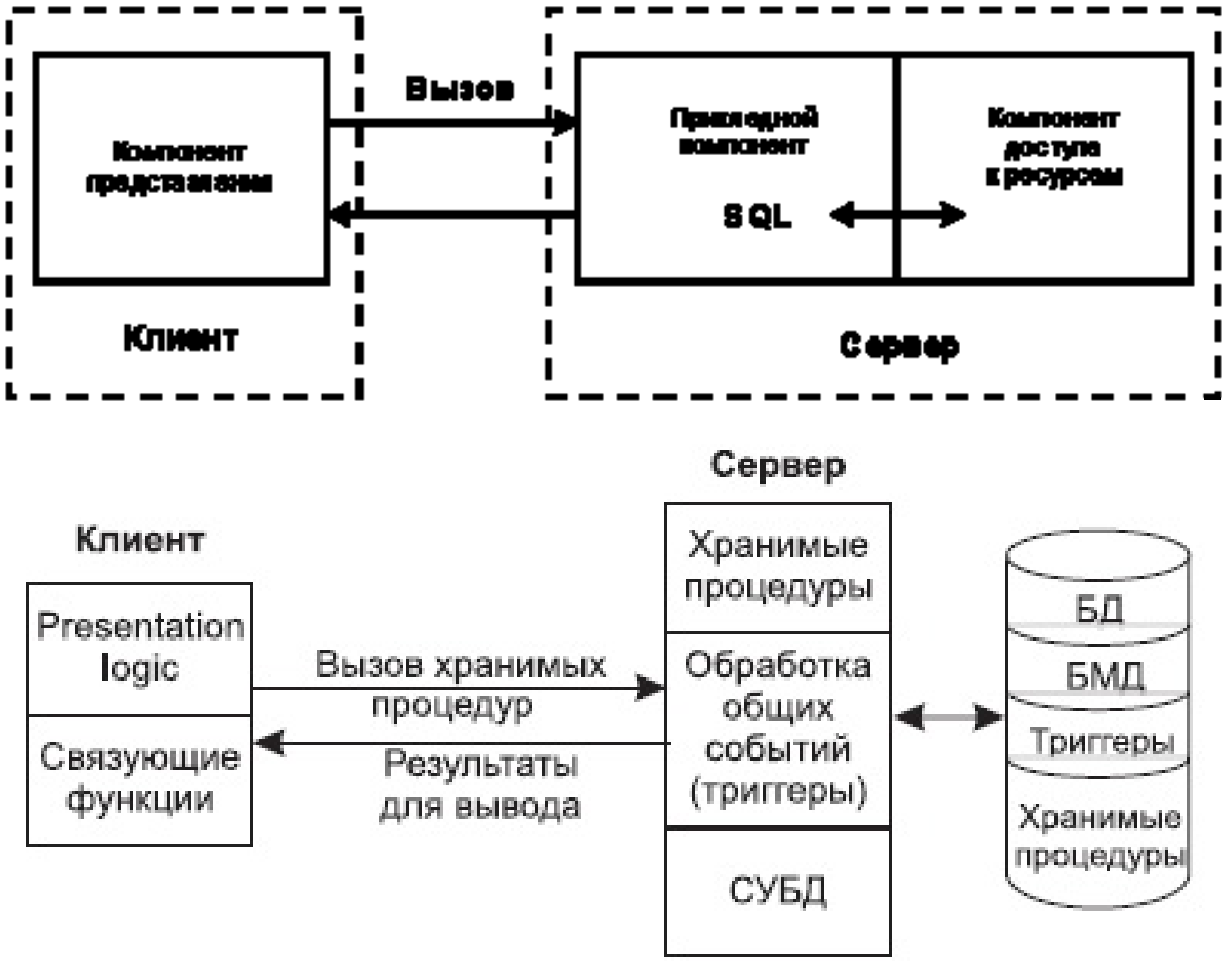


Рис. 7.4. Модель сервера базы данных (два варианта)

В DBS-модели компонент представления выполняется на компьютере-клиенте, в то время как прикладной компонент оформлен как набор хранимых процедур и функционирует на компьютере-сервере БД. Там же выполняется компонент доступа к данным, то есть ядро СУБД.

Достоинства DBS-модели очевидны: это и возможность централизованного администрирования прикладных функций, и снижение трафика (вместо SQL-запросов по сети направляются вызовы хранимых процедур), и возможность разделения процедуры между несколькими приложениями, и экономия ресурсов компьютера за счет использования единожды созданного плана выполнения процедуры.

К недостаткам модели можно отнести ограниченность средств, используемых для написания хранимых процедур, которые представляют собой разнообразные процедурные расширения SQL, не выдерживающие сравнения по изобразительным средствам и функциональным возможностям с языками третьего поколения, такими как C или Pascal. Сфера их использования ограничена конкретной СУБД, в большинстве СУБД отсутствуют возможности отладки и тестирования разработанных хранимых процедур.

Другим недостатком данной модели является очень большая загрузка сервера. Действительно, сервер обслуживает множество клиентов и выполняет следующие функции:

- осуществляет мониторинг событий, связанных с описанными триггерами;
- обеспечивает автоматическое срабатывание триггеров при возникновении связанных с ними событий;
- обеспечивает исполнение внутренней программы каждого триггера;
- запускает хранимые процедуры по запросам пользователей;
- запускает хранимые процедуры из триггеров;
- возвращает требуемые данные клиенту;
- обеспечивает все функции СУБД: доступ к данным, контроль и поддержку целостности данных в БД, контроль доступа, обеспечение корректной параллельной работы всех пользователей с единой БД.

Если мы переложили на сервер большую часть бизнес-логики приложений, то требования к клиентам в этой модели резко уменьшаются. Иногда такую модель называют моделью с «тонким клиентом», в отличие от предыдущих моделей, где на клиента возлагались гораздо более серьезные задачи. Эти модели называются моделями с «толстым клиентом».

На практике часто используется смешанные модели, когда поддержка целостности базы данных и некоторые простейшие прикладные функции обеспечиваются хранимыми процедурами (DBS-модель), а более сложные функции реализуются непосредственно в прикладной программе, которая выполняется на компьютере-клиенте (RDA-модель). Так или иначе современные многопользовательские СУБД опираются на RDA- и DBS-модели и при создании ИС, предполагающем использование только одной СУБД, выбирают одну из этих двух моделей либо их разумное сочетание.

Модель сервера приложений

Эта модель является расширением двухзвенной модели и в ней вводится дополнительное промежуточное звено между клиентом и сервером. Это промежуточное звено содержит один или несколько серверов приложений (рис. 7.5).

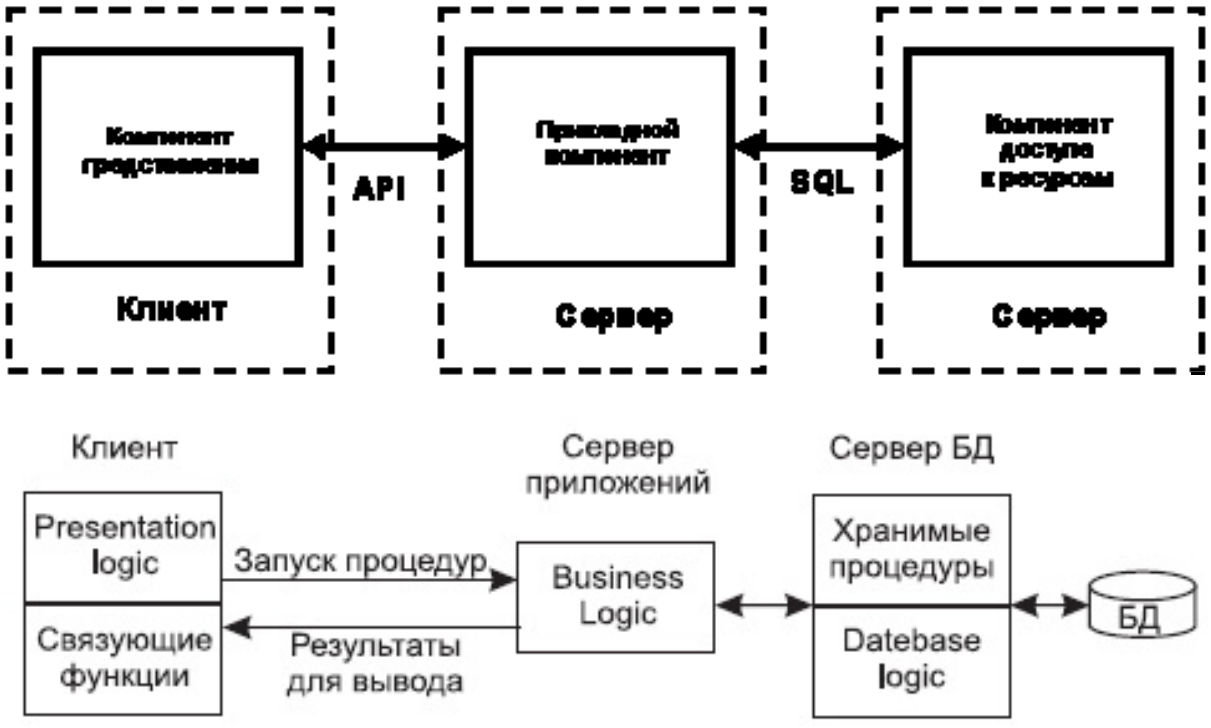


Рис. 7.5. Модель сервера приложений (два варианта)

RDA- и DBS-модели опираются на двухзвенную схему разделения функций. В RDA-модели прикладные функции приданы программе-клиенту, в DBS-модели ответственность за их выполнение берет на себя ядро СУБД. В первом случае прикладной

компонент сливается с компонентом представления, во-втором – интегрируется в компонент доступа к информационным ресурсам. В AS-модели реализована трехзвенная схема разделения функций, где прикладной компонент выделен как важнейший изолированный элемент приложения, для его определения используются универсальные механизмы многозадачной операционной системы и стандартизованы интерфейсы с двумя другими компонентами.

В AS-модели процесс, выполняющийся на компьютере-клиенте, отвечает, как обычно, за интерфейс с пользователем (то есть осуществляет функции первой группы). Обращаясь за выполнением услуг к прикладному компоненту, этот процесс играет роль клиента приложения (Application Client – AC). Прикладной компонент реализован как группа процессов, выполняющих прикладные функции, и называется сервером приложения (Application Server – AS). Все операции над информационными ресурсами выполняются соответствующим компонентом, по отношению к которому AS играет роль клиента. Из прикладных компонентов доступны ресурсы различных типов – базы данных, очереди, почтовые службы и др.

Отметим, что эта модель обладает большей гибкостью, чем двухзвенные модели. Наиболее заметны преимущества модели сервера приложений в тех случаях, когда клиенты выполняют сложные аналитические расчеты над базой данных, которые относятся к области OLAP-приложений. В этой модели большая часть бизнес-логики клиента изолирована от возможностей встроенного SQL, реализованного в конкретной СУБД, и может быть выполнена на стандартных языках программирования. Это повышает переносимость системы, ее масштабируемость.

7.3. Эволюция серверов баз данных

В период создания первых СУБД технология «клиент-сервер» только зарождалась. Поэтому изначально в архитектуре систем не было адекватного механизма организации взаимодействия процессов типа «клиент» и процессов типа «сервер». В современных же СУБД он является фактически основополагающим, и от эффективности его реализации зависит эффективность работы системы в целом.

Рассмотрим эволюцию типов организации подобных механизмов. В основном этот механизм определяется структурой реализации серверных процессов, и часто он называется архитектурой сервера баз данных.

Первоначально существовала модель, когда управление данными (функция сервера) и взаимодействие с пользователем были совмещены в одной программе. Это можно назвать нулевым этапом развития серверов БД (рис. 7.6*а*).

Затем функции управления данными были выделены в самостоятельную группу – сервер, однако модель взаимодействия пользователя с сервером соответствовала парадигме «один-к-одному» (рис. 7.6*б*), то есть сервер обслуживал запросы ровно одного пользователя (клиента) и для обслуживания нескольких клиентов нужно было запустить эквивалентное число серверов.

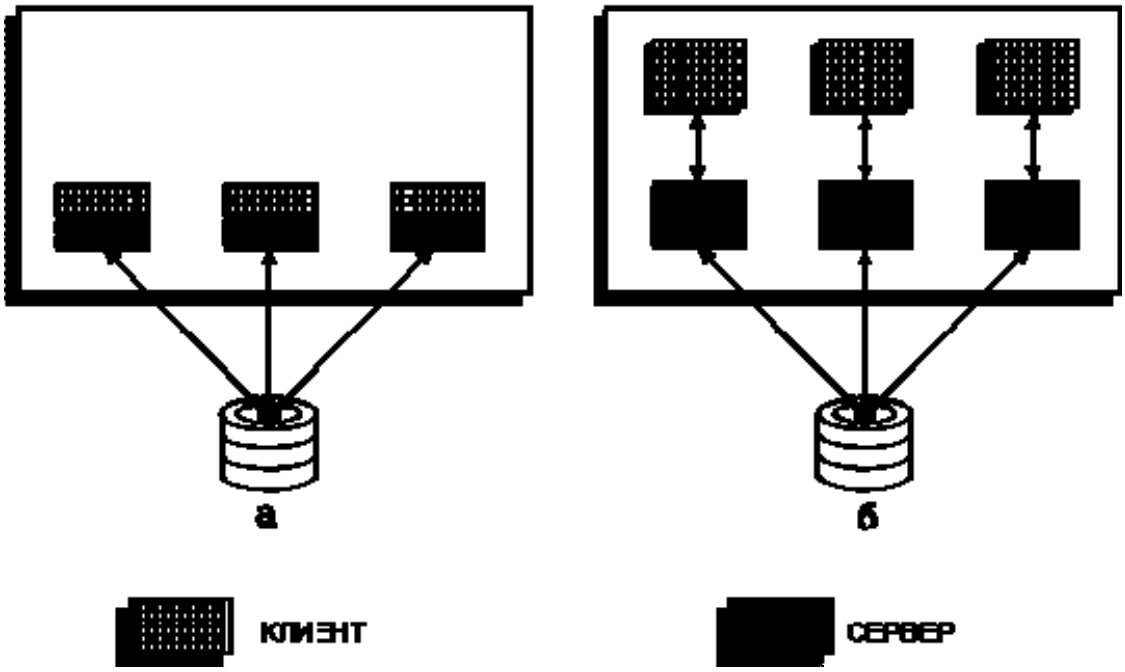


Рис. 7.6. Централизованная архитектура (а)
и архитектура «один к одному» (б)

Выделение сервера в отдельную программу было революционным шагом, который позволил, в частности, поместить сервер на одну машину, а программный интерфейс с пользователем – на другую, осуществляя взаимодействие между ними по сети (рис. 7.7). Однако необходимость запуска большого числа серверов для обслуживания множества пользователей сильно ограничивала возможности такой системы.

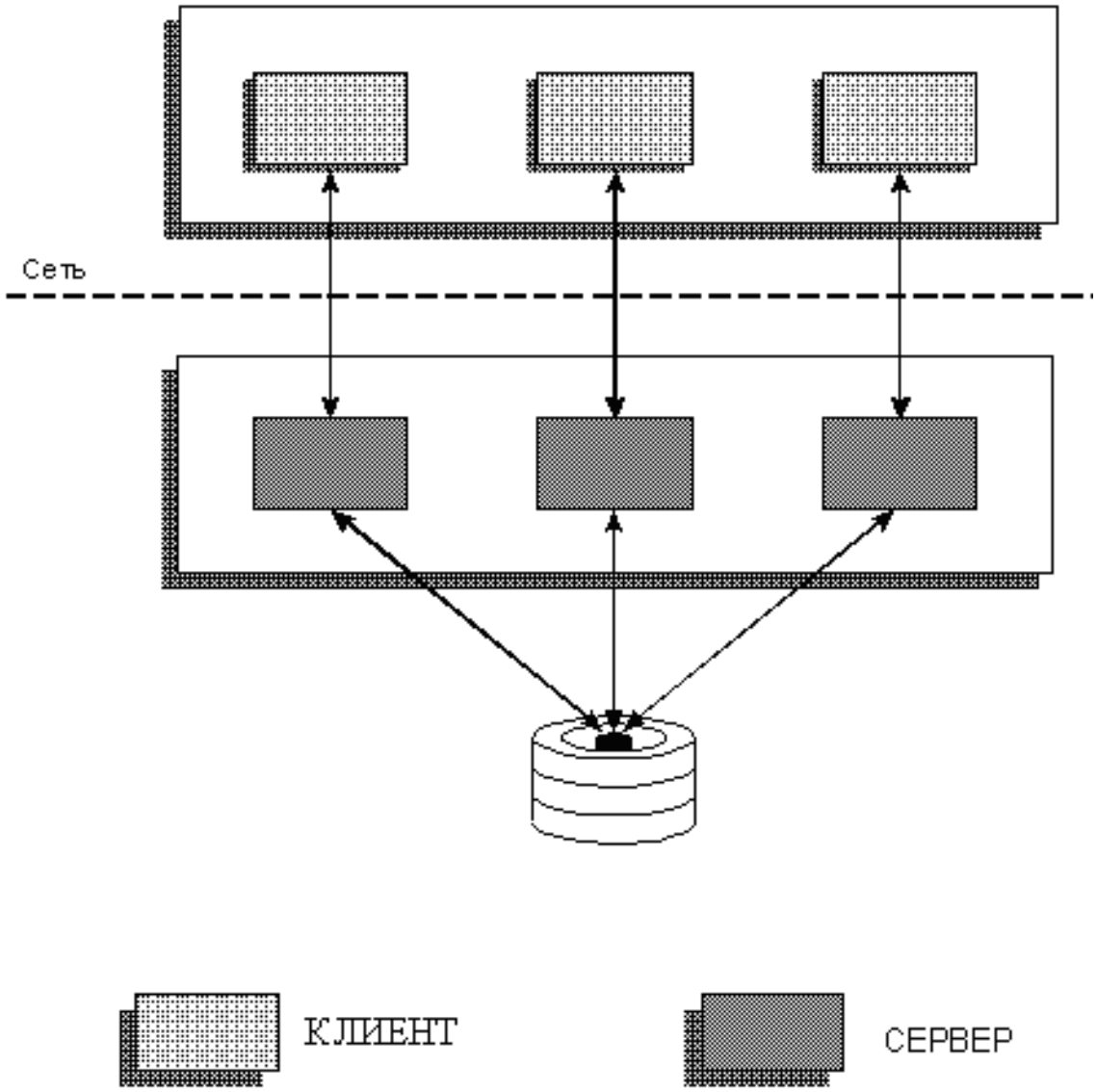


Рис. 7.7. Размещение клиента и сервера на различных машинах

Для обслуживания большого числа клиентов на сервере должно быть запущено большое количество одновременно работающих серверных процессов, а это резко повышало требования к ресурсам ЭВМ, на которой запускались все серверные процессы. Кроме того, каждый серверный процесс в этой модели запускался как независимый, поэтому если один клиент сформировал запрос, который был только что выполнен другим серверным процессом для другого клиента, то запрос, тем не менее, выполнялся повторно. В такой модели весьма сложно обеспечить взаимодействие серверных процессов. Эта модель самая простая, и исторически она появилась первой.

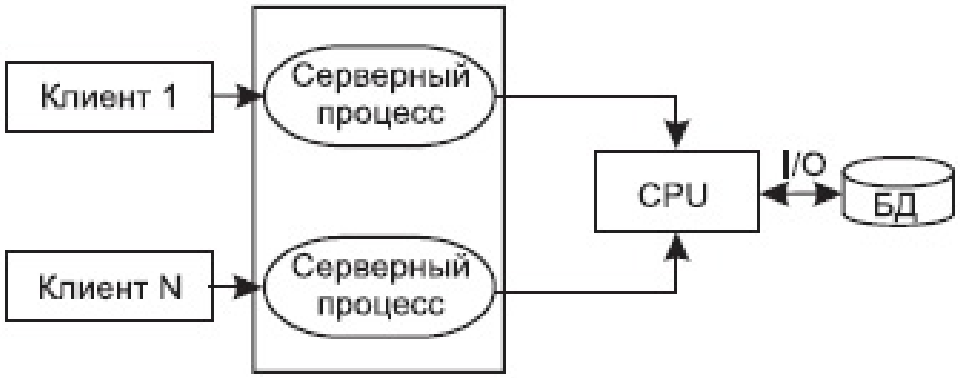


Рис. 7.7а. Размещение клиента и сервера на различных машинах (вариант 2)

Проблемы, возникающие в модели «один-к-одному», решаются в архитектуре «систем с выделенным сервером», который способен обрабатывать запросы от многих клиентов. Сервер единственный обладает монополией на управление данными и взаимодействует одновременно со многими клиентами (рис. 7.8). Логически каждый клиент связан с сервером отдельной нитью или потоком, по которому пересылаются запросы. Такая архитектура получила название многопотоковой односерверной.

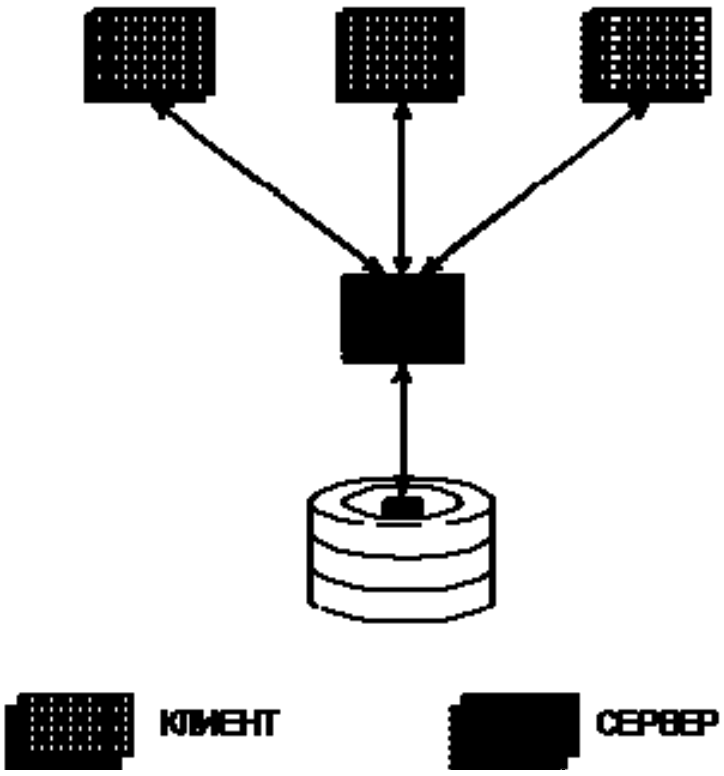


Рис. 7.8. Многопотоковая односерверная архитектура

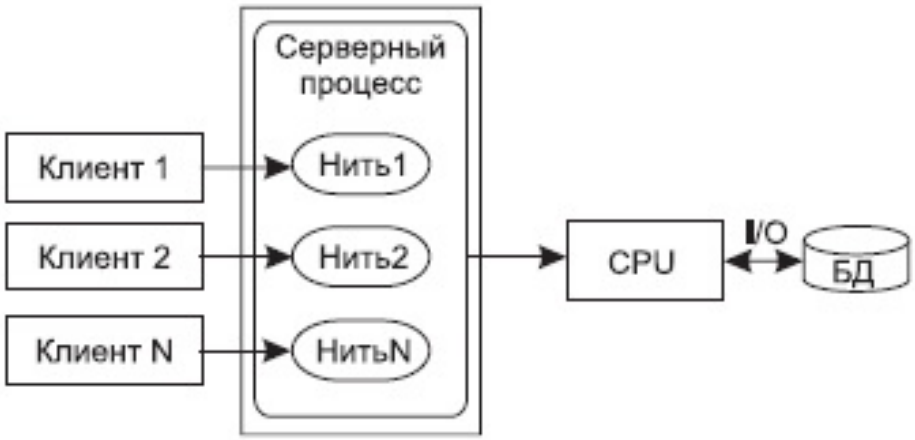


Рис.7. 8а. Многопотоковая односерверная архитектура (вариант 2)

Она позволяет значительно уменьшить нагрузку на операционную систему, возникающую при работе большого числа пользователей. С другой стороны, возможность взаимодействия с одним сервером многих клиентов позволяет в полной степени использовать разделяемые объекты (начиная с открытых файлов и кончая данными из системных каталогов), что значительно уменьшает потребности в памяти и общее число процессов операционной системы. Например, системой с архитектурой «один-к-одному» будет создано 50 копий процессов СУБД для 50 пользователей, тогда как системе с многопотоковой архитектурой для этого понадобится только один серверный процесс.

Однако такое решение тоже имеет свои недостатки. Так как сервер может выполняться только на одном процессоре, возникает естественное ограничение на применение СУБД для мультипроцессорных платформ. Если компьютер имеет, например, четыре процессора, то СУБД с одним сервером используют только один из них, не загружая оставшиеся три.

В некоторых системах эта проблема решается вводом промежуточного диспетчера. Подобная архитектура называется архитектурой виртуального сервера (рис. 7.9).

В этой архитектуре клиенты подключаются не к реальному серверу, а к промежуточному звену, называемому диспетчером, который выполняет только функции диспетчеризации запросов к актуальным серверам, теряя при этом право монопольного распоряжения данными. В этом случае нет ограничений на использование многопроцессорных платформ. Количество актуальных серверов может быть согласовано с количеством процессоров в системе.

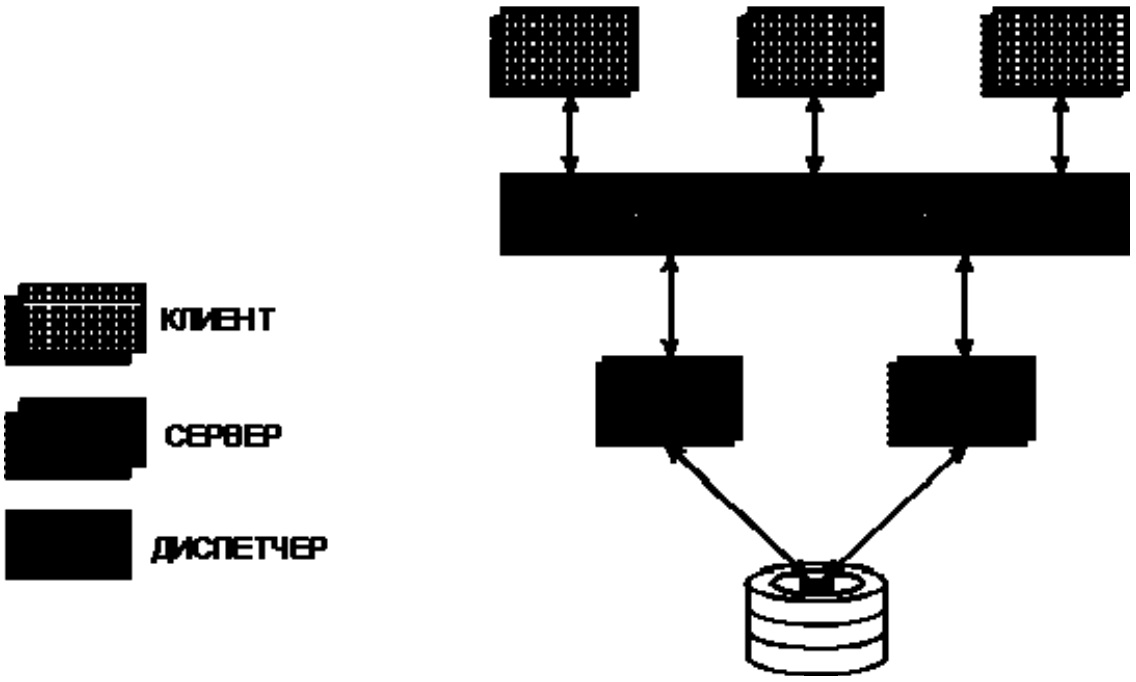


Рис. 7.9. Архитектура с виртуальным сервером

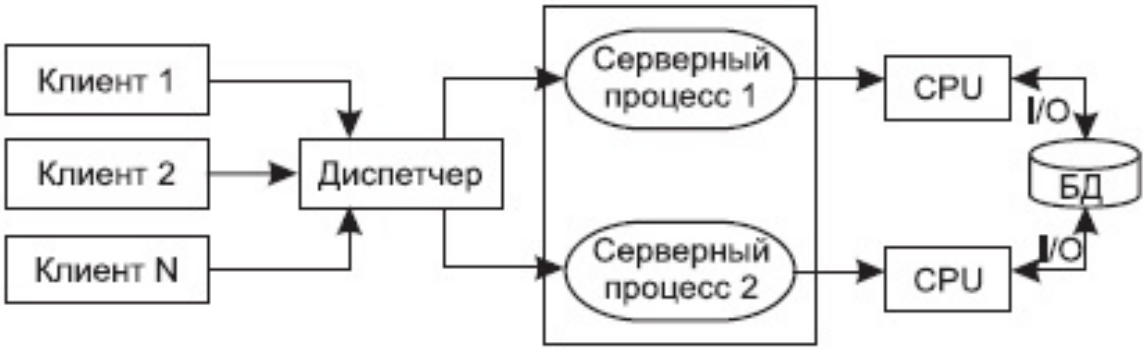


Рис. 7.9а. Архитектура с виртуальным сервером (вариант 2)

Однако и эта архитектура не лишена недостатков, потому что здесь в систему добавляется новый слой, который размещается между клиентом и сервером, что увеличивает трату ресурсов на поддержку баланса загрузки актуальных серверов. Кроме того, введение диспетчера ограничивает возможности управления взаимодействием «клиент-сервер»: во-первых, становится невозможным направить запрос от конкретного клиента конкретному серверу, во-вторых, серверы становятся равноправными – нет возможности устанавливать приоритеты для обслуживания запросов.

Подобная организация взаимодействия «клиент-сервер» является аналогом банка, где имеется несколько окон кассиров, и банковский служащий – администратор зала (диспетчер) –

направляет каждого вновь пришедшего посетителя (клиента) к свободному кассиру (актуальному серверу). Система работает нормально, пока все посетители равноправны (имеют равные приоритеты), однако стоит лишь появиться посетителям с высшим приоритетом, которые должны обслуживаться в специальном окне, как возникают проблемы. Учет приоритета клиентов особенно важен в системах оперативной обработки транзакций, однако именно эту возможность не может предоставить архитектура систем с диспетчеризацией.

Современное решение проблемы СУБД для мультипроцессорных платформ заключается в возможности запуска нескольких серверов базы данных, в том числе и на различных процессорах. При этом каждый из серверов должен быть многопотоковым. Если эти два условия выполнены, то есть основание говорить о многопотоковой архитектуре с несколькими серверами, представленной на рис. 7.10.

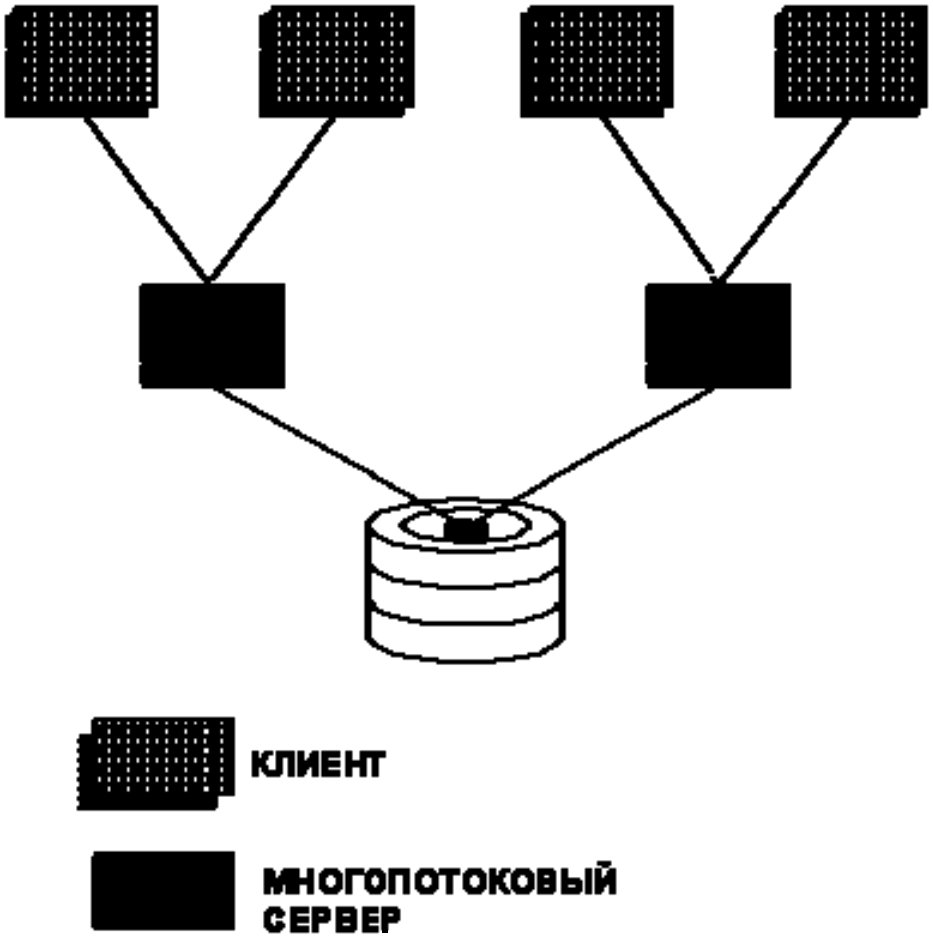


Рис. 7.10. Многопотоковая мультисерверная архитектура

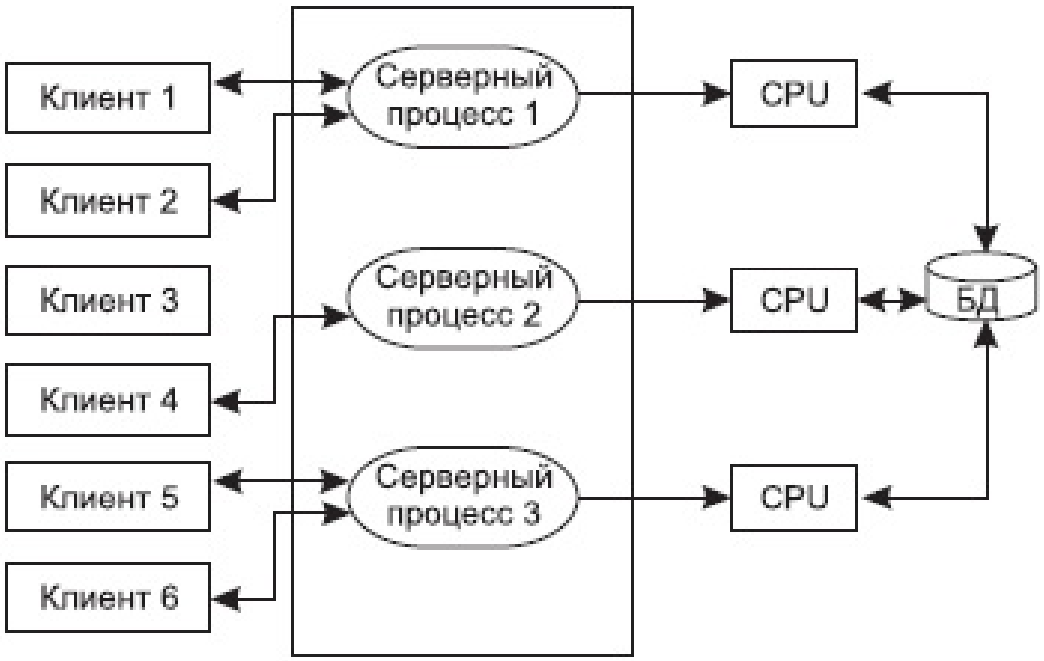


Рис.7.10а. Многопотоковая мультисерверная архитектура (вариант 2)

Эта архитектура может быть связана и с вопросами распараллеливания выполнения одного пользовательского запроса несколькими серверными процессами, и тогда она может быть названа многонитевой мультисерверной архитектурой. Существует несколько возможностей распараллеливания выполнения запроса. В этом случае пользовательский запрос разбивается на ряд подзапросов, которые могут выполняться параллельно, а результаты их выполнения потом объединяются в общий результат выполнения запроса. Тогда для обеспечения оперативности выполнения запросов их подзапросы могут быть направлены отдельным серверным процессам, а потом полученные результаты объединены в общий результат. В данном случае серверные процессы не являются независимыми процессами, такими, как рассматривались ранее. Эти серверные процессы принято называть нитями, и управление нитями множества запросов пользователей требует дополнительных расходов от СУБД, однако при оперативной обработке информации в хранилищах данных такой подход наиболее перспективен.

7.4. Активный сервер

В традиционных моделях взаимодействия «клиент-сервер» (модель файлового сервера, модель доступа к удаленным данным) последнему отводится в основном пассивная роль. Это

означает следующее. Во-первых, сервер базы данных лишен функций хранения и обработки знаний о предметной области. Во-вторых, пассивный сервер не имеет средств контроля за состоянием базы данных и отслеживания событий в базе данных, а также средств программирования реакции на изменения в базе данных, воздействия на работу прикладных программ и возможностей их синхронизации. В-третьих, пассивный сервер не имеет средств обработки нестандартных типов данных.

Знания о предметной области традиционно включались непосредственно в прикладные программы, для чего использовались возможности процедурных языков программирования. Этот подход имеет ряд недостатков.

Реализация правил предметной области (законов, по которым она функционирует) перегружает прикладную программу и усложняет ее написание и понимание основных алгоритмов управления данными. Удобнее оставить за прикладными программами только основные алгоритмы, а часто меняющиеся правила, которые действуют во внешнем мире, вынести за рамки программ и оформить как-то иначе.

Реализация правил предметной области группой разработчиков может привести к противоречивости правил, а разброс правил по многим подпрограммам системы усложняет контроль за взаимным соответствием правил.

Решение задач контроля за состоянием базы данных и уведомления прикладных программ о всех происходящих в ней событиях опирается на механизмы опроса прикладными программами базы данных. Постоянный опрос базы данных сильно сказывается на производительности системы – программы, опрашивающие базу данных, перегружают своими запросами сервер и сеть. Громоздкие конструкции в тексте программ, реализующие опрос, серьезно затрудняют ее написание и понимание.

Обработка новых типов данных (например, единиц различных метрик, таких как футы, дюймы и т. д.) реализуется либо прямым приведением данных новых типов к стандартным, что приводит к потере точности, либо с помощью функций преобразования. Вариант с преобразованием также усложняет конструкцию программы и сказывается на производительности системы. Не понимая нового типа данных, сервер способен лишь послушно сохранять его, не умея сравнивать, сортировать, выполнять

какие бы то ни было операции над данными. Для этого также требуется написание функций. Непонимание нестандартных типов данных делает принципиально невозможным централизованный контроль целостности таких данных.

В СУБД, поддерживающих концепцию активного сервера, знания выносятся за рамки прикладных программ и оформляются как объекты базы данных. Функции применения знаний начинает выполнять непосредственно сервер базы данных. С архитектурой активного сервера связаны следующие механизмы: процедуры базы данных; правила (триггеры); события в базе данных; типы данных, определяемые пользователем.

Проблемы с типами данных, решаются за счет интеграции в сервер **новых типов данных**. К сожалению, далеко не все современные СУБД предоставляют возможность определять собственные типы данных и операции над ними и использовать их в операторах SQL. Введение новых типов данных является по сути изменением ядра СУБД – для этого необходимо написать и добавить в ядро функции обработки данных для определяемого типа.

Определение нового типа данных сводится к указанию в его имени, размера и идентификатора в глобальной структуре, описывающей типы данных. Чтобы с новым типом данных можно было использовать функции, которые реализуют стандартные операции (сравнение, преобразование в различные форматы и т. д.), программист должен разработать их самостоятельно (интерфейс функций предопределен). Указатели на эти функции являются элементами глобальной структуры. Как только новый тип данных определен, то все операции выполняются над ним как над данными стандартного типа. Разрешение пользователю создавать собственные типы данных – один из шагов развития реляционных СУБД в направлении объектно-реляционных систем.

В различных СУБД **процедуры базы данных** носят название хранимых, присоединенных, разделяемых. Процедуры хранятся в словаре БД на сервере в виде специальных программных модулей, реализующих правила предметной области системы, и управляются непосредственно СУБД.

Для написания хранимых процедур используется расширение стандартного языка SQL, так называемый встроенный SQL. Процедура имеет параметры и возвращает значение. Процедура базы данных создается оператором CREATE PROCEDURE и

содержит определения переменных, операторы SQL (например, SELECT, INSERT), операторы проверки условий (IF/THEN/ELSE), операторы цикла (FOR, WHILE), а также некоторые другие.

Клиентское приложение обращается к серверу с командой запуска хранимой процедуры, а сервер выполняет эту процедуру и регистрирует все изменения в БД, которые в ней предусмотрены. Сервер возвращает клиенту данные, соответствующие его запросу, которые требуются либо для вывода на экран, либо для выполнения модулями системы, расположенными на клиенте. Использование процедур базы данных преследует следующие цели.

Во-первых, обеспечивается независимый уровень централизованного контроля доступа к данным, осуществляемый администратором базы данных.

Во-вторых, одна процедура может использоваться несколькими прикладными программами – это позволяет существенно сократить время написания программ за счет оформления их общих частей в виде процедур базы данных. Процедура компилируется и помещается в базу данных, становясь доступной для многократных вызовов. Так как план ее выполнения определяется единожды при компиляции, то при последующих вызовах процедуры фаза оптимизации пропускается, что существенно экономит вычислительные ресурсы системы.

В-третьих, использование процедур баз данных позволяет значительно снизить трафик сети в системах с архитектурой «клиент-сервер». Прикладная программа, вызывающая процедуру, передает серверу лишь ее имя и параметры. В процедуре, как правило, концентрируются повторяющиеся фрагменты из нескольких прикладных программ (рис. 7.11а). Если бы эти фрагменты остались частью программы, они загружали бы сеть посылкой полных SQL-запросов (рис. 7.11б).

В-четвертых, процедуры базы данных (в сочетании с триггерами) предоставляют администратору мощные средства поддержки целостности базы данных.

Механизм **правил (триггеров)** позволяет программировать с помощью встроенного SQL обработку ситуаций, возникающих при любых изменениях в базе данных.

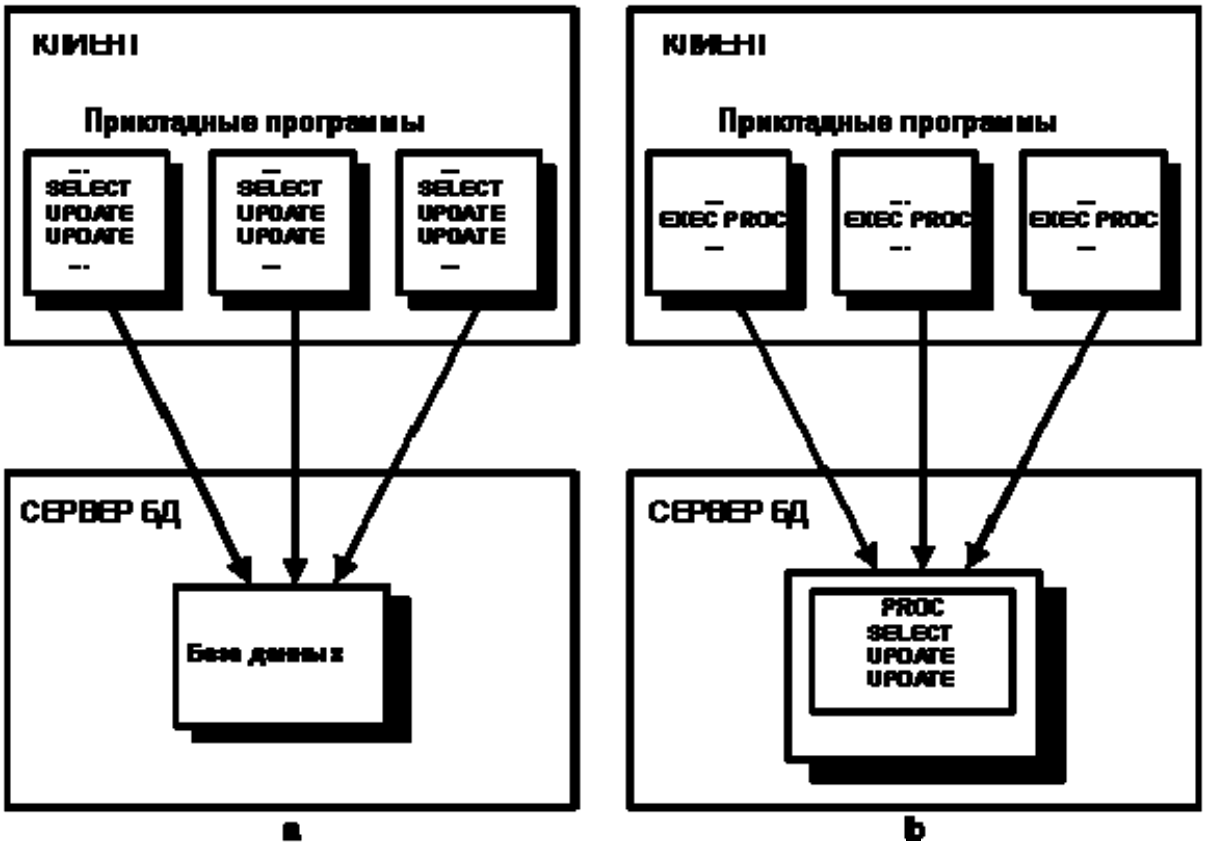


Рис. 7. 11. Увеличение производительности системы за счет использования процедур базы данных: а) процедуры не используются; б) выделение фрагмента прикладных программ в виде процедуры БД

Термин «триггер» взят из электроники и семантически очень точно характеризует механизм отслеживания специальных событий, которые связаны с состоянием БД. Триггер в БД является как бы некоторым тумблером, который срабатывает при возникновении определенного события в БД. Ядро СУБД проводит мониторинг всех событий, которые вызывают созданные и описанные триггеры в БД, и при возникновении соответствующего события сервер запускает соответствующий триггер. Каждый триггер представляет собой также некоторую программу, которая выполняется над базой данных. Триггеры могут вызывать хранимые процедуры.

Правило придается таблице базы данных и применяется при выполнении над ней операций включения, удаления или обновления строк, а также при изменении значений в столбцах таблицы. Применение правила заключается в проверке сформулированных в нем условий, при выполнении которых происходит вызов специфицированной внутри правила процедуры базы

данных. Важно, что правило может быть применено как до, так и после выполнения операции обновления, следовательно, возможна отмена операции. Таким образом, правило позволяет определить реакцию сервера на любое изменение состояния базы данных. Правила (так же, как и хранимые процедуры) хранятся непосредственно в базе данных независимо от прикладных программ.

Использование механизма триггеров преследует следующие цели: обеспечение целостности базы данных и отражение внешних правил деятельности организации.

Данный механизм предполагает, что при срабатывании одного правила могут возникнуть события, которые вызовут срабатывание других триггеров. Этот мощный инструмент требует тонкого и согласованного применения, чтобы не получился бесконечный цикл срабатывания триггеров.

Механизм событий в базе данных позволяет прикладным программам и серверу базы данных уведомлять другие программы о наступлении в базе данных определенного события и тем самым синхронизировать их работу. Операторы языка SQL, обеспечивающие уведомление, часто называют сигнализаторами событий в базе данных. Функции управления событиями целиком ложатся на сервер базы данных. Различные прикладные программы и процедуры вызывают события в базе данных, а сервер оповещает монитор прикладных программ об их наступлении. Реакция монитора на события заключается в выполнении действий, которые предусмотрел его разработчик.

Механизм событий используется следующим образом. Вначале в базе данных для каждого события создается флажок, состояние которого будет оповещать прикладные программы о том, что некоторое событие имело место (оператор CREATE DBEVENT). Далее во все прикладные программы, на ход выполнения которых может повлиять это событие, включается оператор REGISTER DBEVENT, который оповещает сервер базы данных, что данная программа заинтересована в получении сообщения о наступлении события. Теперь любая прикладная программа или процедура базы данных может вызвать событие оператором RAISE DBEVENT. Как только событие произошло, каждая зарегистрированная программа может получить его, для чего должна запросить очередное сообщение из очереди событий (оператор

GET DBEVENT) и запросить информацию о событии, в частности его имя (оператор SQL INQUIRE_SQL).

В заключение рассмотрим пример из производственной системы, иллюстрирующий использование механизма событий в базе данных совместно с правилами и процедурами. Разумеется, пример приведен лишь для иллюстрации схемы срабатывания механизма «правило – процедура – событие» и ни в коей мере не отражает реальные схемы управления технологическими процессами на производстве. События в данном примере используются для определения ситуации, когда рабочий инструмент нагревается до температуры выше допустимой и должен быть отключен.

1. Создается правило, которое применяется всякий раз, когда новое значение температуры инструмента заносится в таблицу Инструмент. Как только она превосходит 500 градусов, правило вызывает процедуру Отключить_инструмент.

```
CREATE RULE Перегрев_инструмента
AFTER UPDATE OF Инструмент (Температура)
WHERE Новое.Температура >= 500
EXECUTE PROCEDURE
    Отключить_инструмент (Номер_инструмента =
  Инструмент.Номер);
```

2. Создается процедура базы данных Отключить_инструмент, которая вызывает событие Перегрев; она будет выполнена в результате применения правила, определенного на шаге 1. Эта процедура регистрирует время, в течение которого инструмент был отключен, и вызывает событие Перегрев:

```
CREATE PROCEDURE Отключить_инструмент
(Номер_инструмента) AS
BEGIN
UPDATE Инструмент
SET Статус = “ВЫКЛ”
WHERE Номер = Номер_инструмента;
RAISE DBEVENT Перегрев;
END;
```

3. Создается событие Перегрев, которое будет вызвано, когда инструмент перегреется:

```
CREATE DBEVENT Перегрев;
```

4. Наконец, создается прикладная программа Монитор Инструментов, которая следит за состоянием инструментов. Она

регистрируется сервером в качестве получателя события Перегрев с помощью оператора REGISTER DBEVENT. Если событие произошло, программа посылает сообщение пользователю и сигнал, необходимый для отключения инструмента.

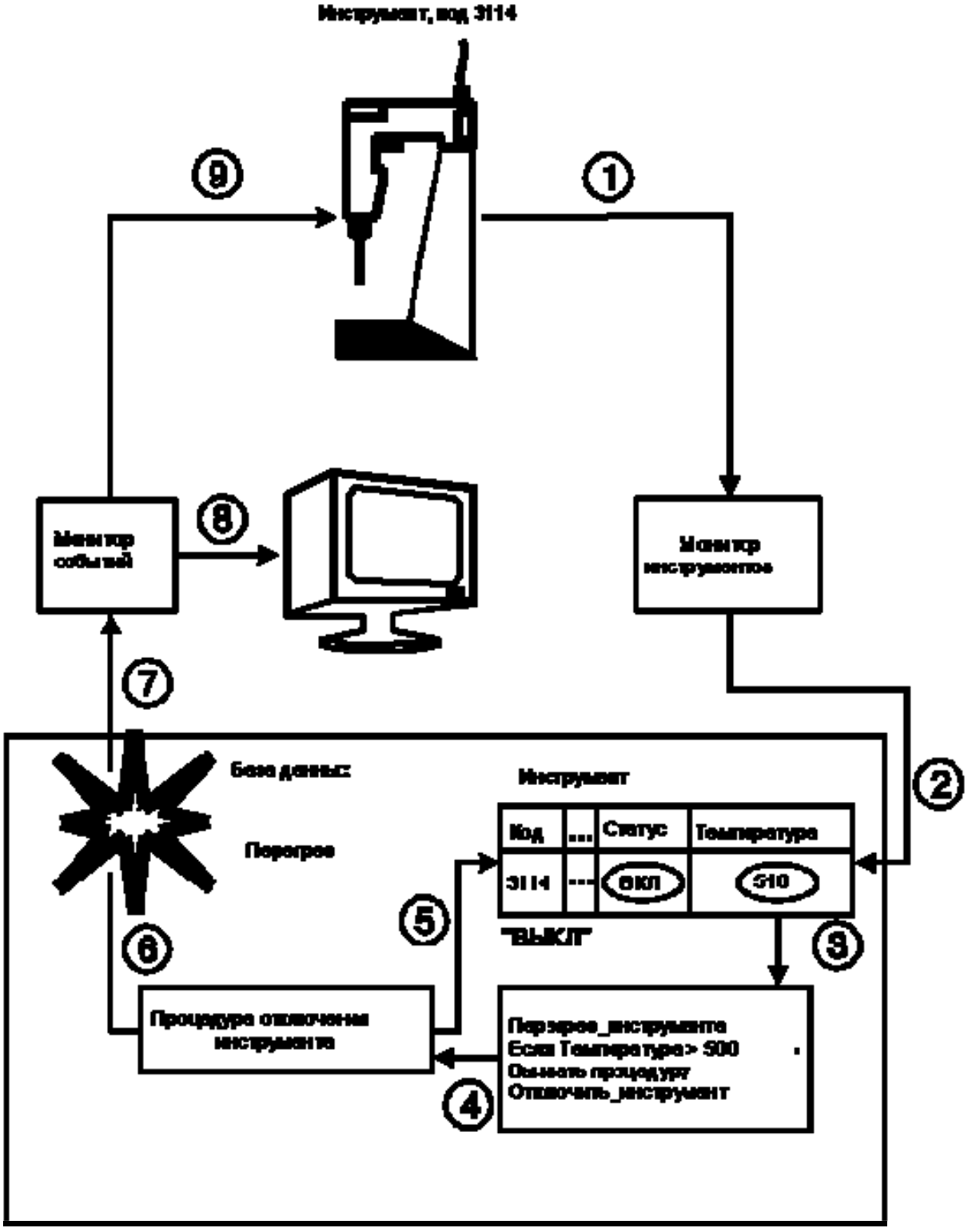


Рис. 7. 12. Пример использования механизма событий в базе данных

EXEC SQL REGISTER
DBEVENT Перегрев;
EXEC SQL GET DBEVENT;
EXEC SQL INQUIRE_SQL **(Имя события = DBEVENTNAME, ...);**
if (Имя события = “Перегрев”)
then
послать сообщение;
отключить инструмент;
endif;

Описанные конструкции в совокупности определяют следующую логику работы:

1. Прикладная программа Монитор Инструментов периодически регистрирует с помощью датчиков текущие значения параметров множества различных инструментов.
2. Та же программа заносит в таблицу Инструмент новое значение температуры для данного инструмента.
3. Всякий раз, когда это происходит, то есть обновляется значение в столбце Температура таблицы Инструмент, применяется правило Перегрев_инструмента.
4. Применение правила состоит в проверке нового значения температуры. Если оно превышает максимально допустимое значение, то запускается процедура Отключить_инструмент.
5. Она изменяет значение в столбце Статус таблицы Инструмент на “ВЫКЛ”.
6. Она же вызывает событие Перегрев.
7. Программа Монитор Событий получает (перехватывает) событие Перегрев.
8. Она же посылает сообщение на экран диспетчеру.
9. Она же отключает инструмент.

Несмотря на то что приведенный пример носит достаточно условный характер, использованная в нем логика универсальна и вполне может быть применена при построении реальных производственных систем.

Контрольные вопросы

1. Опишите основные группы функций стандартного интерактивного приложения.

2. Назовите основные модели клиент-серверных систем. В каких случаях целесообразно применять модель сервера приложений?

3. Что такое активный сервер?

4. Какие средства используются для контроля целостности данных?

Практическое задание

Спроектируйте крупноблочную архитектуру информационной системы университета. Обоснуйте выбор примененной модели и характеристик ее компонентов.

Список литературы

1. Дейт, К. Дж. Введение в системы баз данных / К. Дж. Дейт. – 8-изд. – М.: Вильямс, 2008. – 1328 с.

2. Гарсиа-Молина, Г. Системы баз данных: полный курс / Г. Гарсиа-Молина, Дж. Ульман, Дж. Уидом. – М.; СПб.; Киев: Вильямс, 2003.

3. Пушников, А. Ю. Введение в системы управления базами данных. 1999 / А. Ю. Пушников // CITForum.ru: Центр информационных технологий. – URL: http://citforum.ru/database/dblearn/ (дата обращения: 12.12.2011).

4. Кузнецов, С. Д. Основы баз данных / С. Д. Кузнецов. – 2-е изд. – М.: Бином, 2007. – 484 с.

5. Кузнецов, С.Д. Базы данных: Вводный курс. 2008. / С.Д. Кузнецов. – CITForum.ru: Центр информационных технологий. URL: http://citforum.ru/ database/ advanced_intro/ (дата обращения: 12.12.2011).

6. Коннолли, Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика / Т. Коннолли, К. Бегг. – 3-е изд. – М.: Вильямс, 2003. – 1440 с.

7. Марков, А. С. Базы данных: учебник / А. С. Марков, К. Ю. Лисовский. – М.: Финансы и статистика, 2004. – 512 с.

8. Грофф, Дж. Р. SQL: Полный справочник / Дж Р. Грофф, П. Н. Вайнберг, Э. Дж. Оппель. – 3-е изд. – М.: Вильямс, 2011. – 960 с.

9. Кригель, А. SQL. Библия пользователя / А. Кригель, Б. Трухнов. – 2-е изд. – М.: Вильямс, 2010. – 752 с.

10. Кириллов, В. В. Введение в реляционные базы данных / В. В. Кириллов, Г. Ю. Громов. – СПб.: BHV, 2009. – 464 с.

11. Карпова, Т. С. Базы данных: модели, разработка, реализация: учебник / Т. С. Карпова. – СПб.: Питер, 2001. – 304 с.

12. SQL.ru : все про SQL, базы данных, программирование и разработку информационных систем. – URL: http://sql.ru (дата обращения: 12.12.2011).

Учебное издание

Зафиевский Александр Владимирович
Короткин Алексей Абрамович
Лататуев Александр Николаевич

Базы данных

Учебное пособие

Редактор, корректор М. В. Никулина
Верстка И. Н. Иванова

Подписано в печать 15.03.12. Формат 60×84 1/16.
Бум. офсетная. Гарнитура «Times New Roman».
Усл. печ. л. 9,30. Уч.-изд. л. 7,54.
Тираж 50 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета им. П. Г. Демидова.
Отпечатано на ризографе.
Ярославский государственный университет им. П. Г. Демидова.
150000, Ярославль, ул. Советская, 14.

А. В. Зафиевский
А. А. Короткин
А. Н. Лататуев

Базы данных