ЧАСТЬ I
БАЗЫ ДАННЫХ, СУБД И МОДЕЛИ ДАННЫХ

Глава 1
НАЗНАЧЕНИЕ ТЕХНОЛОГИИ БАЗ ДАННЫХ. ФУНКЦИИ И ОСНОВНЫЕ КОМПОНЕНТЫ СИСТЕМ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

В данной главе определяется понятие информационной системы, обсуждаются предпосылки появления в компьютерах устройств внешней памяти, а также обосновывается принципиальная важность для организации информационных систем дисковых устройств с подвижными магнитными головками. Далее рассматриваются особенности организации и основное функциональное назначение одного из ключевых компонентов современных операционных систем — систем управления файлами. В третьем подразделе главы демонстрируется, что возможности файловых систем оказываются недостаточными для создания информационных программных систем и то, что естественные требования информационных систем к средствам управления данными во внешней памяти приводят к необходимости наличия систем управления базами данных (СУБД). В ходе этого анализа определяются основные функции, которые должны поддерживаться СУБД, и основные компоненты, которые содержит типичная СУБД.

1.1. Информационные системы и устройства внешней памяти

В общем смысле информационная система представляет собой программный комплекс, функции которого состоят в поддержке надежного долговременного хранения информации в памяти компьютера, выполнении требуемых для данного приложения преобразований информации и/или вычислений, предостав-

лении пользователям системы удобного и легко осваиваемого интерфейса. Обычно объемы данных, с которыми приходится иметь дело таким системам, достаточно велики, а сами данные обладают достаточно сложной структурой. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

Надежное и долговременное хранение информации можно обеспечить только при наличии запоминающих устройств, сохраняющих информацию после выключения электропитания. Оперативная (основная) память этим свойством обычно не обладает. В первые десятилетия развития вычислительной техники использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. Емкость магнитных лент была достаточно велика, но по своей природе они обеспечивали только последовательный доступ к данным. Емкость магнитной ленты пропорциональна ее длине. Чтобы получить доступ к требуемой порции данных, нужно в среднем перемотать половину ее длины. Но чисто механическую операцию перемотки нельзя выполнить очень быстро. Поэтому быстрый произвольный доступ к данным на магнитной ленте, очевидно, невозможен.

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

Указанные ограничения не очень существенны для систем численных расчетов. Обсудим более подробно, какие реальные потребности возникают у разработчиков таких систем. Прежде всего, для получения требуемых результатов серьезные вычислительные программы должны проработать достаточно долгое время (недели, месяцы и даже, может быть, годы). Наличие гарантий надежности со стороны производителей аппаратных компьютерных средств не избавляет программистов от необходимости использования программного сохранения частичных результатов вычислений, чтобы при возникновении непредвиденных сбоев аппаратуры можно было продолжить выполнение расчетов с некоторой контрольной точки. Для сохранения проме-

жуточных результатов идеально подходят магнитные ленты: при выполнении процедуры установки контрольной точки данные последовательно сбрасываются на ленту, а при необходимости перезапуска от сохраненной контрольной точки данные также последовательно с ленты считываются.

Вторая традиционная потребность программистов вычислительных приложений — максимально большой объем оперативной памяти. Большая оперативная память требуется, во-первых, для того, чтобы обеспечить программе быстрый доступ к большому количеству обрабатываемых данных. Во-вторых, сложные вычислительные программы сами могут иметь большой объем. Поскольку объем реально доступной в ЭВМ оперативной памяти всегда являлся недостаточным для удовлетворения текущих потребностей вычислений, требовалась быстрая внешняя память для организации оверлеев и/или виртуальной памяти. Здесь не будем вдаваться в детали организации этих механизмов программного расширения оперативной памяти, но заметим, что для этого идеально подходили магнитные барабаны. Они обеспечивали быстрый доступ к внешней памяти, а для расширения оперативной памяти одной программы (сложные вычислительные программы, как правило, выполняются на компьютере в одиночку) большой объем внешней памяти не требуется.

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

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

Именно требования к устройствам внешней памяти со стороны бизнес-приложений вызвали появление устройств внешней

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

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

При выполнении обмена с диском аппаратура выполняет три основных действия:

- подвод головок к нужному цилиндру (обозначим время выполнения этого действия как $t_{\text{пг}}$);
- поиск на дорожке нужного блока (время выполнения — $t_{\text{пб}}$);
- собственно обмен с этим блоком (время выполнения — $t_{\text{об}}$).

Тогда, как правило, $t_{\text{пг}} \gg t_{\text{пб}} \gg t_{\text{об}}$, потому что подвод головок — это механическое действие, причем в среднем нужно переместить головки на расстояние, равное половине радиуса

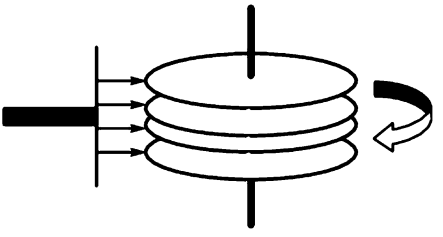

Рис. 1.1. Грубая схема дискового устройства памяти с подвижными головками

поверхности, а скорость передвижения головок не может быть слишком большой по физическим соображениям. Поиск блока на дорожке требует прокручивания пакета магнитных дисков в среднем на половину длины внешней окружности; скорость вращения диска может быть существенно больше скорости движения головок, но она тоже ограничена законами физики. Для выполнения же обмена нужно прокрутить пакет дисков всего лишь на угловое расстояние, соответствующее размеру блока. Таким образом, из всех этих действий в среднем наибольшее время занимает первое, и поэтому существенный выигрыш в суммарном времени обмена при считывании или записи только части блока получить практически невозможно.

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

1.2. Файловые системы

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

В этом подразделе будут обсуждаться история файловых систем, их основные черты и области разумного применения. Однако сначала уместно сделать два замечания. Во-первых, заметим, что в области управления файлами исторически

существует некоторая терминологическая путаница. Термин *файловая система* (*file system*) используется для обозначения как программной системы, управляющей файлами, так и архива файлов, хранящегося во внешней памяти. Было бы лучше в первом случае использовать термин *система управления файлами*, оставив за термином *файловая система* только второе значение. Однако принятая практика вынуждает использовать в этой книге термин *файловая система* в обоих смыслах. Точный смысл термина должен быть понятен из контекста. (Аналогичная путаница возникает при некорректном использовании терминов *база данных* и *система управления базами данных*. В данной книге эти термины строго различаются.) Во-вторых, для целей этой главы достаточно ограничиться описанием основных свойств так называемых традиционных файловых систем, не затрагивая особенности современных систем с повышенной надежностью.

Первая развитая файловая система была разработана специалистами IBM в середине 1960-х гг. для выпускавшейся компанией серии компьютеров System/360. В этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы (а также файлы с прямым доступом к записям), а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами. Контроллеры обеспечивали возможность обмена с дисковыми устройствами порциями данных произвольного размера, а также индексный доступ к записям файлов, и эти функции контроллеров активно использовались в файловой системе OS/360.

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

1.2.1. Структуры файлов

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

Во-первых, считывание или запись только части блока не приводит к существенному выигрышу в суммарном времени обмена, поскольку, как указывалось в подразд. 1.1, основная часть этого времени тратится на перемещение магнитных головок. Во-вторых, для работы с частями блоков файловая система должна обеспечить буферы оперативной памяти соответствующего размера, что существенно усложняет распределение основной памяти. Алгоритмы распределения памяти порциями произвольного размера плохи тем, что любой из них рано или поздно приводит к *внешней фрагментации* памяти. В памяти образуется большое число мелких свободных фрагментов. Их совокупный размер может быть больше размера любого требуемого буфера, но его можно выделить, только если произвести сжатие памяти, т.е. подвижку всех занятых фрагментов таким образом, чтобы они располагались вплотную один к другому. Во время выполнения операции сжатия памяти нужно приостановить выполнение обменов, а сама эта операция занимает много времени.

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

В некоторых файловых системах базовый уровень был доступен пользователю, но чаще он прикрывался некоторым более

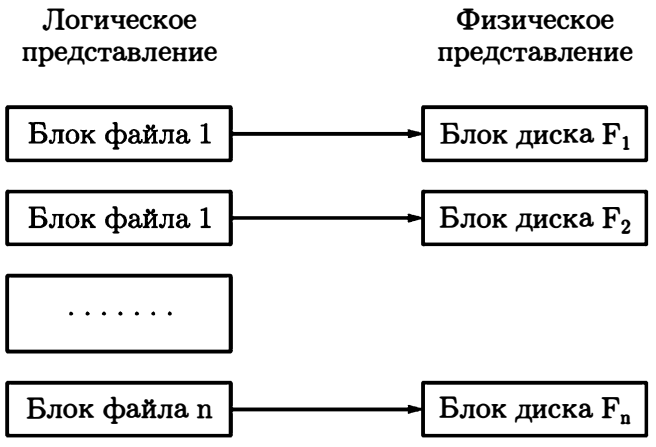

Рис. 1.2. Схематичное изображение базового файла

высоким уровнем, стандартным для пользователей. Исторически существует два основных подхода. При первом подходе, свойственном, например, файловой системе операционной системы компании Hewlett-Packard OpenVMS, пользователи представляют файл как последовательность записей. Каждая запись — это последовательность байтов, имеющая постоянный или переменный размер. Можно читать или писать записи последовательно либо позиционировать файл на запись с указанным номером.

В некоторых файловых системах допускается структуризация записей на поля и объявление указываемых полей ключами записи. В таких файловых системах можно потребовать выборку записи из файла по ее заданному ключу. Естественно, в этом случае файловая система поддерживает в том же (или другом, служебном) базовом файле дополнительные, невидимые пользователю, служебные структуры данных — *индексы*. Распространенные способы организации индексов ключевых файлов основаны на технике *хэширования* и *B-деревьев* (подробно эта техника обсуждается в гл. 8 в контексте физической организации хранения баз данных). Существуют и многоключевые способы организации файлов (у одного файла объявляется несколько ключей, и можно выбирать записи по значению каждого ключа).

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

Конечно, в обоих случаях можно обеспечить набор преобразующих функций, приводящих представление файла к другому виду. Примером тому может служить поддержка стандартной файловой среды UNIX в среде операционной системы OpenVMS.

1.2.2. Логическая структура файловых систем и именование файлов

Во всех современных файловых системах обеспечивается многоуровневое именование файлов за счет наличия во внешней памяти каталогов — дополнительных файлов со специальной

структурой. Каждый каталог содержит имена каталогов и/или файлов, хранящихся в данном каталоге. Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл.

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

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

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

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

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

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

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

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

выполнении стартового командного файла), пользователи ОС UNIX, как правило, и не задумываются о происхождении общей файловой системы.

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

1.2.3. Авторизация доступа к файлам

Поскольку файловая система является общим хранилищем файлов, принадлежащих, вообще говоря, разным пользователям, системы управления файлами должны обеспечивать *авторизацию* доступа к файлам. В общем виде подход состоит в том, что для каждого зарегистрированного пользователя и для каждого существующего файла файловой системы указываются действия, выполнение которых над данным файлом разрешено или запрещено данному пользователю (так называемый мандатный способ защиты — у каждого пользователя имеется или не имеется отдельный мандат для работы с каждым файлом). Применение мандатного способа авторизации доступа влечет за собой существенные накладные расходы, связанные с потребностью хранения избыточной информации и использованием этой информации для проверки правомочности доступа.

Поэтому в большинстве современных систем управления файлами применяется упрощенный подход к *авторизации* доступа к файлам, традиционно поддерживаемый, например в ОС UNIX (так называемый *дискреционный* подход). При использовании этого подхода с каждым зарегистрированным пользователем связывается пара целочисленных идентификаторов: идентификатор группы, к которой относится пользователь, и его собственный идентификатор. Этими же идентификаторами снабжается каждый процесс, запущенный от имени данного пользователя и имеющий возможность обращаться к системным вызовам файловой системы. Соответственно, при каждом файле хранится полный идентификатор пользователя (собственный идентификатор плюс идентификатор группы), который создал этот файл, и помечается, какие действия с файлом может производить он сам, какие действия с файлом доступны для остальных пользователей той же группы и что могут делать с файлом пользователи других групп. Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Хранимая информация очень компактна (два целых

числа для представления идентификаторов и шкала из 9 бит для характеристики возможных действий), при проверке требуется небольшое количество действий, и этот способ контроля доступа в большинстве случаев удовлетворителен.

1.2.4. Синхронизация многопользовательского доступа

Если операционная система поддерживает многопользовательский режим, может возникнуть ситуация, когда два или более пользователей (процессов) одновременно пытаются работать с одним и тем же файлом. Если все эти пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этой группы требуется взаимная синхронизация.

В файловых системах обычно применяется следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывается режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторого процесса A файл уже открыт некоторым другим процессом B, причем существующий режим открытия несовместим с требуемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы либо процессу A сообщается о невозможности открытия файла в нужном режиме, либо процесс A блокируется до тех пор, пока процесс B не выполнит операцию закрытия файла.

1.2.5. Области разумного применения файлов

После краткого обзора технологии файловых систем обсудим возможные области их применения. Чаще всего файлы используются для хранения текстовых данных: документов, текстов программ и т. д. Такие файлы обычно создаются и модифицируются с помощью различных текстовых редакторов. Эти редакторы могут быть очень простыми, такими, как *ed* в мире UNIX или утилиты редактирования (*Far Manager*, *WordPad* и т. д.) в среде Windows. Они могут быть сложными и многофункциональными, синтаксически ориентированными, как, например, *GNU Emacs*. Но обычно структура текстовых файлов очень проста (с точки зрения файловой системы): это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки). Конечно же, сложность логической

структуры текстового файла определяется текстовым редактором, но в любом случае файловой системе она не видна.

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

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

Одним словом, файловые системы обычно обеспечивают хранение данных с очень «аморфной» стандартной структурой, оставляя дальнейшую структуризацию прикладным программам. В перечисленных ранее случаях использования файлов это

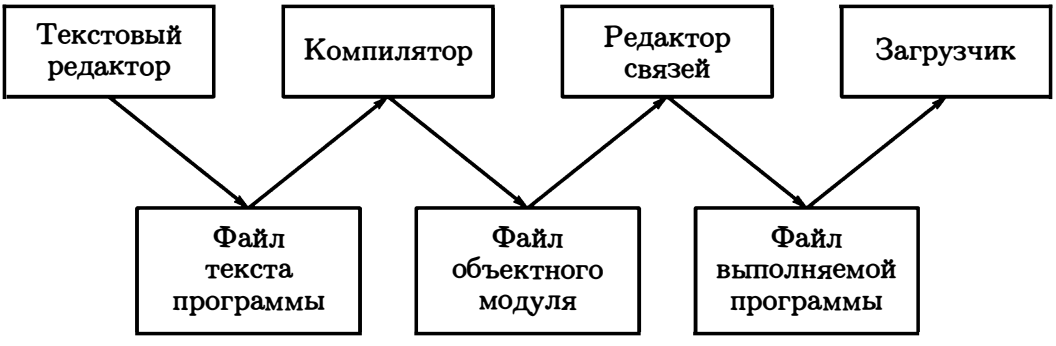

Рис. 1.3. Связи между программными компонентами по пониманию логической структуры файлов

даже хорошо, поскольку при разработке любой новой прикладной системы, опираясь на простые, стандартные и сравнительно дешевые средства файловой системы, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.

1.3. Потребности информационных систем

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

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

Но поскольку для функционирования информационных систем требуются сложные структуры данных, эти дополнительные индивидуальные средства управления данными являлись существенной частью информационных систем и практически повторялись от одной системы к другой. Стремление выделить общую часть информационных систем, ответственную за управление сложно структурированными данными, явилось, на мой взгляд, первой побудительной причиной создания СУБД. Очень скоро стало понятно, что невозможно обойтись общей библиотекой программ (рис. 1.5), реализующей над стандартной базовой файловой системой более сложные методы хранения данных.

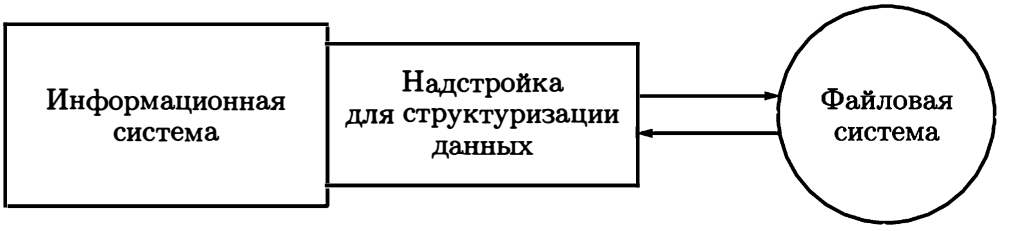

Рис. 1.4. Примитивная схема структуризации данных в информационной системе

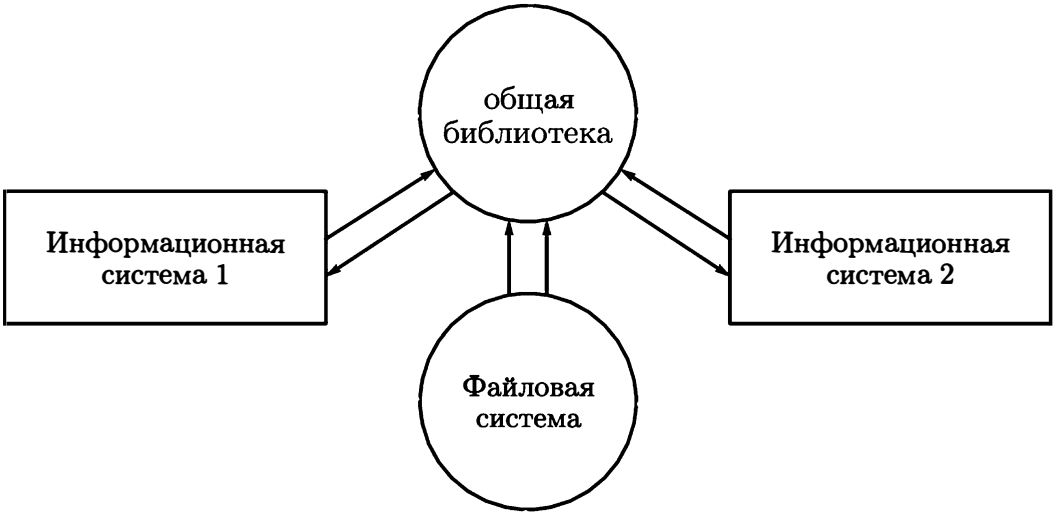

Рис. 1.5. Две информационные системы с общей библиотекой

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

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

Кроме того, для каждого отдела должна поддерживаться возможность получения:

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

Для каждого служащего должна поддерживаться возможность получения:

- номера удостоверения по полному имени служащего (для простоты допустим, что имена всех служащих различны);
- полного имени по номеру удостоверения;
- информации о соответствии служащего занимаемой должности и размере его заработной платы.

1.3.1. Структуры данных

Предположим, что мы решили основывать эту информационную систему на файловой системе и пользоваться одним файлом

СЛУЖАЩИЕ, расширив базовые возможности файловой системы за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является служащий, в этом файле должна содержаться одна запись для каждого служащего. Чтобы можно было удовлетворить указанные ранее требования, запись о служащем должна иметь следующие поля:

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

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

Чтобы информационная система могла эффективно выполнять свои базовые функции, необходимо обеспечить многоключевой доступ к файлу СЛУЖАЩИЕ по уникальным ключам (ключ называется уникальным, если его значения гарантированно различны во всех записях файла) СЛУ_ИМЯ и СЛУ_НОМЕР. Очевидно, что в противном случае для выполнения наиболее часто используемых операций получения данных о конкретном служащем понадобится последовательный просмотр в среднем половины записей файла.

Кроме того, должна обеспечиваться возможность эффективного выбора всех записей с общим значением СЛУ_ОТД_НОМЕР, т.е. доступ по неуникальному ключу. Если не поддерживать специальный механизм доступа, то для получения данных об отделе в целом в общем случае потребуется полный просмотр файла. Требуемая общая структура файла СЛУЖАЩИЕ показана на рис. 1.6. Но даже в этом случае, чтобы получить численность отдела или общий размер заработной платы, система должна будет выбрать все записи о служащих указанного отдела и посчитать соответствующие общие значения.

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

1) требуется создание достаточно сложной надстройки для многоключевого доступа к файлам;

2) возникает существенная избыточность данных (для каждого служащего повторяется имя руководителя его отдела);

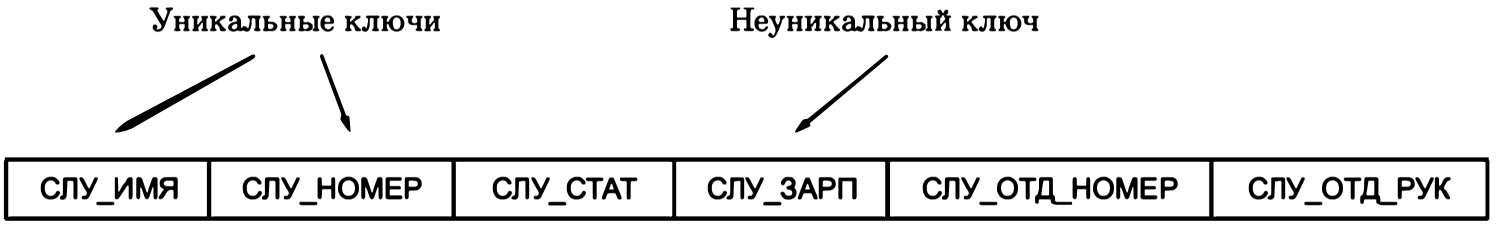

Рис. 1.6. Структура файла СЛУЖАЩИЕ на уровне приложения (случай одного файла)

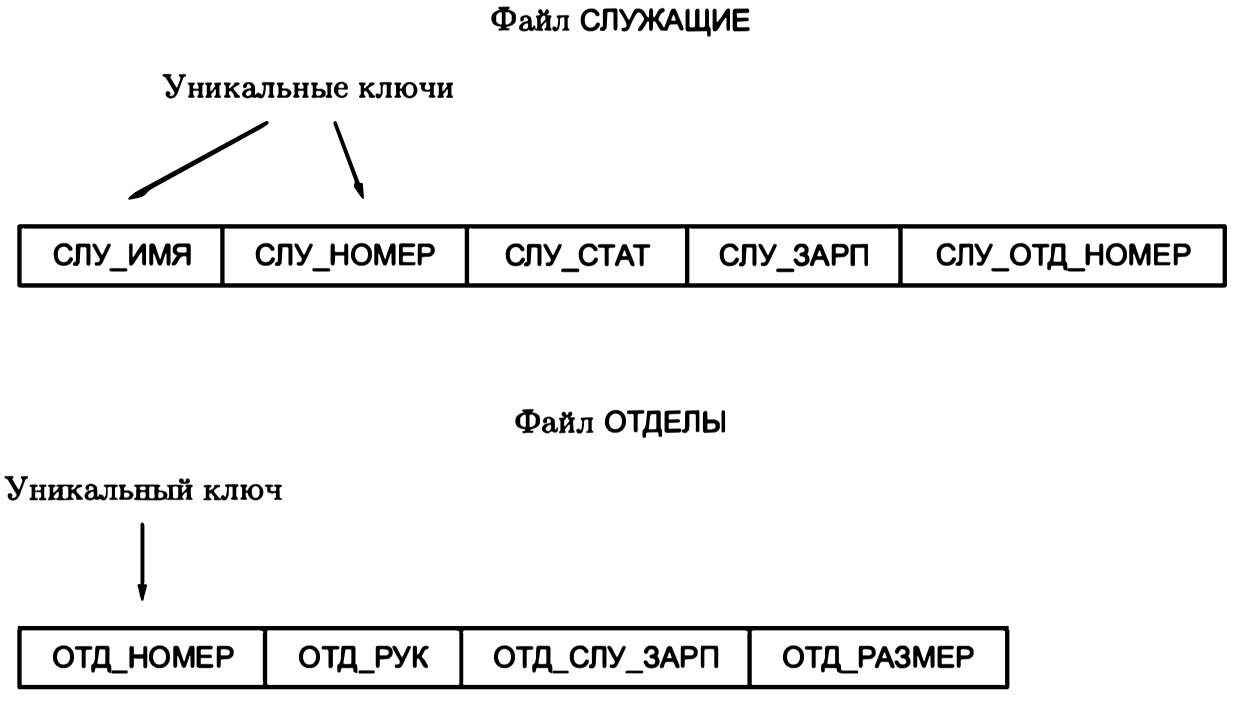

Рис. 1.7. Структура файлов СЛУЖАЩИЕ и ОТДЕЛЫ на уровне приложения (случай двух файлов)

3) требуется выполнение массовой выборки и вычислений для получения суммарной информации об отделах.

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

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

Введение этих двух файлов позволило бы преодолеть большинство неудобств, перечисленных в предыдущем абзаце. Каждый из файлов содержал бы только не дублируемую информацию, не возникала бы необходимость в динамических вычислениях суммарной информации по отделам. Но заметим, что при таком переходе наша информационная система должна обладать некоторыми новыми особенностями, сближающими ее с СУБД.

1.3.2. Целостность данных

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

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

(1) если в файле СЛУЖАЩИЕ содержится запись со значением поля СЛУ_ОТД_НОМЕР, равным n, то и в файле ОТДЕЛЫ долж-

на содержаться запись со значением поля ОТД_НОМЕР, также равным *n*;

(2) если в файле ОТДЕЛЫ содержится запись со значением поля ОТД_РУК, равным *m*, то и в файле СЛУЖАЩИЕ должна содержаться запись со значением поля СЛУ_НОМЕР, также равным *m*; далее в книге мы увидим, что правила (1) и (2) являются частными случаями общего правила *ссылочной целостности:* поле СЛУ_ОТД_НОМЕР содержит «ссылки» на записи таблицы ОТДЕЛЫ, и поле ОТД_РУК содержит «ссылки» на записи таблицы СЛУЖАЩИЕ;

(3) при любом корректном состоянии информационной системы значение поля ОТД_СЛУ_ЗАРП любой записи *отд_k* файла ОТДЕЛЫ должно быть равно сумме значений поля СЛУ_ЗАРП всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СЛУ_ОТД_НОМЕР совпадает со значением поля ОТД_НОМЕР записи *отд_k*;

(4) при любом корректном состоянии информационной системы значение поля ОТД_РАЗМЕР любой записи *отд_k* файла ОТДЕЛЫ должно быть равно числу всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СЛУ_ОТД_НОМЕР совпадает со значением поля ОТД_НОМЕР записи *отд_k*; далее будет показано, что правила (3) и (4) представляют собой примеры общих ограничений целостности базы данных.

Понятие согласованности, или целостности, данных является ключевым понятием баз данных. Фактически, если в информационной системе (даже такой простой, как в рассматриваемом примере) поддерживается согласованное хранение данных в нескольких файлах, можно говорить о том, что в ней поддерживается база данных. Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных.

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

1.3.3. Языки запросов

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

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

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

Возможна и другая формулировка того же запроса (*запрос 2*):

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

Это пример запроса на языке SQL с *вложенным подзапросом*. Во вложенном подзапросе выбирается значение поля **СЛУ_ОТД_НОМЕР** из записи файла **СЛУЖАЩИЕ**, в которой значение поля **СЛУ_ИМЯ** равняется строковой константе **'ПЕТР ИВАНОВИЧ СИДОРОВ'**. Если такая запись существует, то она единственная,

поскольку поле СЛУ_ИМЯ является уникальным ключом файла СЛУЖАЩИЕ. Тогда результатом выполнения подзапроса будет единственное значение — номер отдела, в котором работает Петр Иванович Сидоров. Во внешнем запросе это значение будет ключом доступа к файлу ОТДЕЛЫ, и снова будет выбрана только одна запись, поскольку поле ОТД_НОМЕР является уникальным ключом файла ОТДЕЛЫ. Если же на данном предприятии Петр Иванович Сидоров не работает, то подзапрос выдаст пустой результат и внешний запрос тоже выдаст пустой результат.

Приведенные примеры показывают, что при формулировке запроса с использованием SQL можно не задумываться о том, как будет выполняться этот запрос. Среди метаданных БД будет содержаться информация о том, что поле СЛУ_ИМЯ является ключевым для файла СЛУЖАЩИЕ (т.е. по заданному значению имени служащего можно быстро найти соответствующую запись или убедиться в том, что запись с таким значением поля СЛУ_ИМЯ в файле отсутствует), а поле ОТД_НОМЕР — ключевое для файла ОТДЕЛЫ (и более того, оба ключа в соответствующих файлах являются уникальными), и система сама воспользуется этим. Можно формально доказать, что формулировки *запроса 1* и *запроса 2* эквивалентны, т.е. вне зависимости от состояния данных всегда производят один и тот же результат. Наиболее вероятным способом выполнения запроса в обеих формулировках будет выборка записи из файла СЛУЖАЩИЕ со значением поля СЛУ_ИМЯ, равным строке 'ПЕТР ИВАНОВИЧ СИДОРОВ', взятие из этой записи значения поля СЛУ_ОТД_НОМЕР и выборка из таблицы ОТДЕЛЫ записи с таким же значением поля ОТД_НОМ.

При возникновении потребности в получении списка сотрудников, не соответствующих занимаемой должности, достаточно обратиться к системе с запросом (*запрос 3*):

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

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

1.3.4. Транзакции, журнализация и многопользовательский режим

Представим себе, что в первоначальной реализации информационной системы, основанной на использовании библиотек

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

Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии (точно будут нарушены правила (3) и (4), а может быть, и правила (1) и (2), сформулированные в подразд. 1.3.2). Потребуется выяснить это (а для этого нужно явно проверить соответствие данных в файлах СЛУЖАЩИЕ и ОТДЕЛЫ) и привести данные в согласованное состояние. Проверку и коррекцию можно выполнить, например, следующим образом. Сгруппировать записи файла СЛУЖАЩИЕ по значениям поля СЛУ_ОТД_НОМЕР. Для каждой группы (a) проверить, существует ли в файле ОТДЕЛЫ запись, значение поля ОТД_НОМ которой равняется значению поля СЛУ_ОТД_НОМЕР записей данной группы; если такой записи в файле ОТДЕЛЫ нет, то (b) исключить группу из файла СЛУЖАЩИЕ и перейти к обработке следующей группы; иначе (c) посчитать число записей в группе и вычислить суммарное значение заработной платы; (d) обновить полученными значениями поля ОТД_РАЗМЕР и ОТД_СЛУ_ЗАРП соответствующей записи файла ОТДЕЛЫ и перейти к обработке следующей группы.

Настоящие СУБД берут такую работу на себя, поддерживая транзакционное управление и журнализацию изменений базы данных (см. подразд. 1.4). Прикладная система не обязана заботиться о поддержке корректности состояния БД, хотя и должна «знать», какие цепочки операций изменения данных являются допустимыми.

Представим теперь, что в информационной системе требуется обеспечить параллельную (например, многотерминальную) работу с базой данных служащих и отделов. Если опираться только на использование файлов, то для обеспечения корректности на все время модификации любого из двух файлов доступ других пользователей к этому файлу будет блокирован (вспомните возможности файловых систем в отношении синхронизации параллельного доступа, упоминавшиеся в подразд. 1.2.5). Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о служащем Иване Сидоровиче Петрове, даже если они работают в разных

отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

1.4. Основные функции и компоненты СУБД

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

1.4.1. СУБД как независимый системный компонент

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

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

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

Вариант 2. Можно скопировать метаданные системы регистрации служащих в бухгалтерскую систему. Но метаданные (как и данные) не обязательно являются статичными. Структура

* Это замечание применимо, по крайней мере, к СУБД *общего назначения*, поддерживающей все возможности, рассмотренные в подразд. 1.3. Встраиваемые *специализированные* СУБД достаточно распространены.

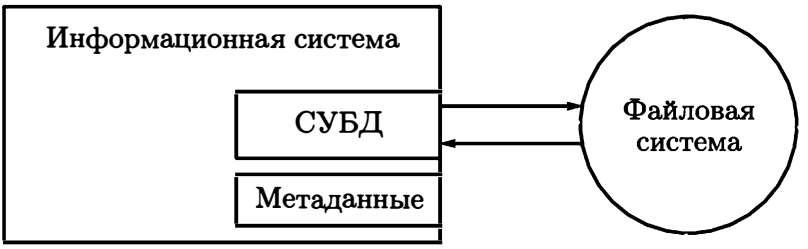

Рис. 1.8. СУБД в составе информационной системы

базы данных может со временем изменяться, могут исчезать одни правила целостности и появляться другие. Как согласовывать копии метаданных, поддерживаемые независимыми информационными системами? Так мы приходим к организации системы, показанной на рис. 1.9, где можно видеть три информационные системы, которые через одну СУБД работают с двумя разными базами данных, причем первая и вторая системы работают с общей базой данных. Это возможно, поскольку метаданные каждой базы данных содержатся в самих базах данных, и достаточно лишь указать СУБД, с какой базой данных желает работать данное приложение. Поскольку СУБД функционирует отдельно от приложений и ее работа с базами данных регулируется метаданными, совместное использование одной базы данных двумя информационными системами не вызовет потери согласованности данных, и доступ к данным будет должным образом синхронизироваться. Заметим, что рис. 1.9

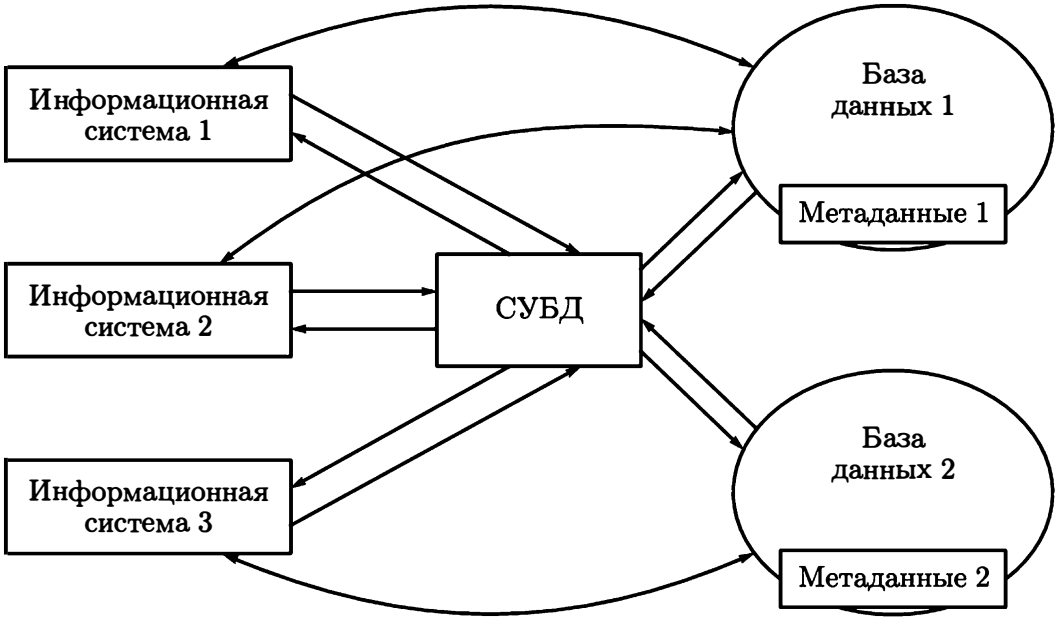

Рис. 1.9. Отдельная СУБД и базы данных с метаданными

вплотную приближает нас к наиболее распространенной в последние десятилетия архитектуре «клиент-сервер». СУБД играет роль «сервера», обсуживающего нескольких «клиентов» — прикладных информационных систем.

1.4.2. Функции СУБД

В подразд. 1.3 было выявлено несколько потребностей информационных систем, которые не покрываются возможностями систем управления файлами: поддержка логически согласованного набора файлов; восстановление согласованного состояния данных после разного рода сбоев; обеспечение высокого уровня параллелизма работы нескольких пользователей; поддержка языка манипулирования данными. Эти и другие функции традиционно поддерживаются СУБД. В этом подразделе обсудим функции СУБД более строго.

Структуризация данных во внешней памяти. Эта функция обеспечивает *поддержку необходимых структур данных во внешней памяти* как для хранения данных и матаданных, непосредственно входящих в БД, так и для служебных целей, например, для ускорения доступа к данным в некоторых случаях (обычно для этого используются индексы). В некоторых реализациях СУБД активно используются возможности существующих файловых систем, в других работа производится на уровне устройств внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД файловую систему, и, если использует, то как организованы файлы. В частности, в СУБД обычно поддерживается собственная система именования объектов БД.

Управление буферами оперативной памяти. СУБД обычно работают с БД значительного размера; по крайней мере, этот размер обычно существенно больше объема доступной оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является *буферизация данных в оперативной памяти.* Даже если операционная система производит общесистемную буферизацию данных (как, например, в случае ОС UNIX), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части БД. Поэтому в развитых СУБД поддерживается собственный набор буферов

оперативной памяти с использованием собственной дисциплины замены буферов*.

Управление транзакциями. *Транзакция* — это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно завершается и СУБД фиксирует изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример информационной системы с файлами СЛУЖАЩИЕ и ОТДЕЛЫ, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового служащего является объединение элементарных операций над файлами СЛУЖАЩИЕ и ОТДЕЛЫ в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД. Но понятие транзакции гораздо более важно в многопользовательских СУБД.

То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к БД. При соответствующем управлении посредством СУБД параллельно выполняемыми транзакциями каждый пользователь может в принципе ощущать себя единственным пользователем СУБД (на самом деле, это несколько идеализированное представление, поскольку в некоторых случаях пользователи многопользовательских СУБД могут ощутить присутствие своих коллег).

С управлением транзакциями в многопользовательской СУБД связаны важные понятия *сериализации транзакций*. Под сериализацией параллельно выполняемых транзакций понимается такой порядок планирования выполнения операций, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Понятно, что если удается добиться действительно сериализованного выполнения

---

* Следует оговориться, что существует класс СУБД, ориентированный на работу с базами данных, постоянно и полностью размещаемыми в основной памяти (Main-Memory DBMS). В этих системах, естественно, отсутствует компонент управления буферами и вообще многое устроено по-другому. Однако, по крайней мере, пока эти системы играют лишь вспомогательную роль при управлении данными, и в этой книге их особенности не обсуждаются.

смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. Наиболее распространены алгоритмы, основанные на синхронизационных блокировках объектов БД. При использовании любого алгоритма сериализации возможны конфликты между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержки сериализации необходимо выполнить *откат* (ликвидировать все изменения, произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

Журнализация. Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризуемые потерей информации на носителях внешней памяти. Примерами программных сбоев могут быть аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной. Первую ситуацию можно рассматривать как особый вид мягкого аппаратного сбоя; при возникновении последней требуется ликвидировать последствия только одной транзакции.

Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, для обеспечения надежного хранения данных в БД требуется хранение избыточных данных, причем та часть данных, которая используется для восстановления БД, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение *журнала изменений* БД.

Журнал — это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью (например,

можно поддерживать две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. В разных СУБД изменения БД журнализуются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения БД (например, операции удаления строки из таблицы реляционной БД), иногда — минимальной внутренней операции модификации страницы внешней памяти; в некоторых системах одновременно используются оба подхода.

Во всех случаях принято придерживаться стратегии «упреждающей» записи в журнал (так называемого протокола Write Ahead Log — WAL). Грубо говоря, эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части БД. Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Поддержка языков БД. Для работы с базами данных используются специальные языки, в целом называемые *языками баз данных*. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка — *язык определения данных* (*DDL — Data Definition Language*) и *язык манипулирования данными* (*DML — Data Manipulation Language*). При этом язык DDL служил, главным образом, для определения логической структуры БД, т. е. той структуры БД, какой она представляется пользователям; язык DML содержал набор операторов манипулирования данными, т. е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Более подробно языки ранних СУБД будут рассмотрены в гл. 2.

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (*Standard Query Language*). В нескольких главах этой книги язык SQL будет рассматриваться достаточно подробно, а пока мы перечислим основные функции реляционной СУБД, поддерживаемые на «языковом» уровне (т. е. функции, поддерживаемые при реализации интерфейса SQL).

Прежде всего, язык SQL сочетает средства языков DDL и DML, т. е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД — в основном, именование таблиц и их столбцов) поддерживается на языковом уровне в том смысле, что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых служебных таблиц-каталогов. Внутренняя часть СУБД (ядро) вообще не работает с именами таблиц и их столбцов.

Язык SQL содержит специальные средства определения ограничений целостности БД. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т. е. при первичной обработке операторов модификации БД компилятор SQL на основании имеющихся в БД ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые представления БД, фактически являющиеся хранимыми в БД запросами (результатом любого запроса к реляционной БД является таблица) с именованными столбцами. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в БД, но с помощью представлений можно ограничить или, наоборот, расширить видение БД для конкретного пользователя. Поддержание представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам БД производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу БД, обладает полным набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

1.4.3. Типовая организация современной СУБД

Организация типичной СУБД и состав ее компонентов соответствуют рассмотренному ранее набору функций. Напомним, что мы выделили следующие основные функции СУБД:

- управление данными во внешней памяти;
- управление буферами оперативной памяти;
- управление транзакциями;
- журнализация и восстановление БД после сбоев;
- поддержка языков БД.

Логически в современной реляционной СУБД можно выделить внутреннюю часть — ядро СУБД (часто его называют Data Base Engine), компилятор языка БД (например, SQL), подсистему поддержки времени выполнения, набор утилит. В некоторых системах эти части выделяются явно, в других — нет, но логически такое разделение можно провести во всех СУБД.

Ядро СУБД отвечает за управление данными во внешней памяти, управление буферами оперативной памяти, управление транзакциями и журнализацию. Соответственно, можно выделить такие компоненты ядра (по крайней мере, логически, хотя в некоторых системах эти компоненты выделяются явно), как менеджер данных, менеджер буферов, менеджер транзакций и менеджер журнала. Как можно понять из предшествующего материала этой главы, функции этих компонентов взаимосвязаны, и для обеспечения корректной работы СУБД все эти компоненты должны взаимодействовать по тщательно продуманным и проверенным протоколам. Ядро СУБД обладает собственным интерфейсом, обычно недоступным пользователям напрямую и используемым в программах, производимых компилятором SQL (или в подсистеме поддержки выполнения таких программ) и утилитах БД. Ядро СУБД является основной резидентной частью СУБД. При использовании архитектуры «клиент-сервер» ядро является базовой составляющей серверной части системы.

Главной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Существенной проблемой реляционных СУБД является то, что языки этих систем (а это, как правило, SQL) являются непроцедурными, т. е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия. Поэтому компилятор должен решить, каким образом выполнять оператор языка, прежде чем произвести программу. Применяются достаточно сложные методы оптимизации операторов. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах, но более часто в выполняемом внутреннем машинно-независимом коде. В последнем случае реальное вы-

полнение оператора производится с привлечением подсистемы поддержки времени выполнения, представляющей собой, по сути дела, интерпретатор этого внутреннего языка.

Наконец, в отдельные утилиты БД обычно выделяют такие процедуры, которые слишком накладно выполнять с использованием языка БД, например, загрузка и выгрузка БД, сбор статистики, глобальная проверка целостности БД и т. д. Утилиты программируются с использованием интерфейса ядра СУБД, а иногда даже с проникновением внутрь ядра.

* * *

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

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

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

При выборе технологии построения информационной системы нужно тщательно оценивать и прогнозировать ее потенциальные потребности в средствах управления данными. Конечно, любую информационную систему можно основывать на использовании промышленной, большой и мощной СУБД. Но вполне может

оказаться так, что в действительности приложение будет использовать доли процентов общих возможностей СУБД. Накладные расходы (затраты на дополнительную аппаратуру, лицензирование дорогостоящего программного продукта, увеличение общего времени выполнения операций) могут оказаться неоправданными.

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