ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современные авторы часто употребляют термины «банк данных» и «база данных» как синонимы, однако в общеотраслевых руководящих материалах по созданию банков данных Государственного комитета по науке и технике (ГКНТ), изданных в 1982 г., эти понятия различаются. Там приводятся следующие определения банка данных, базы данных и СУБД:

Банк данных (БнД) – это система специальным образом организованных данных – баз данных, программных, технических, языковых, организационно-методических средств, предназначенных для обеспечения централизованного накопления и коллективного многоцелевого использования данных.

База данных (БД) – именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области.

Система управления базами данных (СУБД) – совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями.

Эти определения четко разграничивают назначение всех трех базовых понятий, и мы можем принять их за основу.

Программы, с помощью которых пользователи работают с базой данных, называются приложениями. В общем случае с одной базой данных могут работать множество различных приложений. Например, если база данных моделирует некоторое предприятие, то для работы с ней может быть создано приложение, которое обслуживает подсистему учета кадров, другое приложение может быть посвящено работе подсистемы расчета заработной платы сотрудников, третье приложение работает как подсистемы складского учета, четвертое приложение посвящено планированию производственного процесса. При рассмотрении приложений, работающих с одной базой данных, предполагается, что они могут работать параллельно и независимо друг от друга, и именно СУБД призвана обеспечить работу множества приложений с единой базой данных таким образом, чтобы каждое из них выполнялось корректно, но учитывало все изменения в базе данных, вносимые другими приложениями.

Архитектура базы данных. В процессе научных исследований, посвященных тому, как именно должна быть устроена СУБД, предлагались различные способы реализации. Самым жизнеспособным из них оказалась предложенная американским комитетом по стандартизации ANSI (American National Standards Institute) трехуровневая система организации БД, изображенная на рисунке 1.1.

Уровень внешних моделей – самый верхний уровень, где каждая модель имеет свое «видение» данных. Этот уровень определяет точку зрения на БД отдельных приложений. Каждое приложение видит и обрабатывает

только те данные, которые необходимы именно этому приложению. Например, система распределения работ использует сведения о квалификации сотрудника, но ее не интересуют сведения об окладе, домашнем адресе и телефоне сотрудника, и наоборот, именно эти сведения используются в подсистеме отдела кадров.

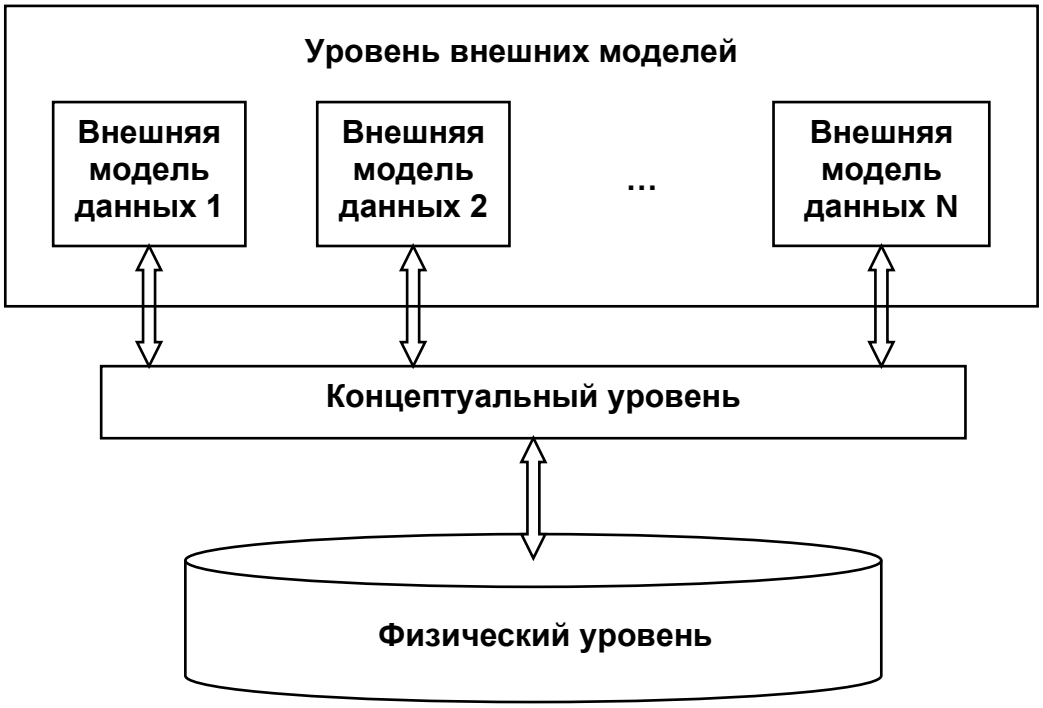


Рис. 1.1. Архитектура базы данных

Концептуальный уровень – центральное управляющее звено, здесь база данных представлена в наиболее общем виде, который объединяет данные, используемые всеми приложениями, работающими с данной базой данных. Фактически концептуальный уровень отражает обобщенную модель предметной области (объектов реального мира), для которой создавалась база данных. Как любая модель, концептуальная модель отражает только существенные, с точки зрения обработки, особенности объектов реального мира. Выделение концептуального уровня позволило разработать аппарат централизованного управления базой данных.

Физический уровень – собственно данные в файлах или в страничных структурах, расположенных на внешних носителях информации.

Эта архитектура позволяет обеспечить логическую (между уровнями 1 и 2) и физическую (между уровнями 2 и 3) независимость при работе с данными.

Логическая независимость предполагает возможность изменения одного приложения без корректировки других приложений, работающих с этой же базой данных.

Физическая независимость предполагает возможность переноса хранимой информации с одних носителей на другие при сохранении работоспособности всех приложений, работающих с данной базой данных. Это именно то, чего не хватало при использовании файловых систем.

Уровни моделей данных. Объединяя частные представления о содержимом базы данных, полученные в результате опроса пользователей, и свои представления о данных, которые могут потребоваться в будущих приложениях, администратор базы данных (АБД) сначала создает обобщенное неформальное описание создаваемой базы данных. Это описание, выполненное с использованием естественного языка, математических формул, таблиц, графиков и других средств, понятных всем людям, работающих над проектированием базы данных, называют инфологической моделью данных.

Рис. 1.2. Уровни моделей данных

Инфологическая модель отображает реальный мир в некоторые понятные человеку концепции, полностью независимые от параметров среды хранения данных. Такая человеко-ориентированная модель полностью независима от физических параметров среды хранения данных. В конце концов, этой средой может быть память человека, а не ЭВМ. Поэтому инфологическая модель не должна изменяться до тех пор, пока какие-то изменения в реальном мире не потребуют изменения в ней некоторого определения, чтобы эта модель продолжала отражать предметную область.

Существует множество подходов к построению таких моделей: графовые модели, семантические сети, модель «сущность-связь» и т.д. Наиболее популярной из них оказалась модель «сущность-связь» (ER-модель), которая будет рассмотрена в разделе 2.

Остальные модели данных, показанные на рисунке 1.2, являются компьютеро-ориентированными. С их помощью СУБД дает возможность программам и пользователям осуществлять доступ к хранимым данным лишь по их именам, не заботясь о физическом расположении этих данных. Нужные данные отыскиваются СУБД на внешних запоминающих устройствах по физической модели данных.

Так как указанный доступ осуществляется с помощью конкретной СУБД, то модели должны быть описаны на языке описания данных этой СУБД. Такое описание, создаваемое АБД по инфологической модели данных, называют *даталогической моделью* данных.

Инфологическая модель должна быть отображена в компьютеро-ориентированную даталогическую модель, «понятную» СУБД. В процессе развития теории и практического использования баз данных, а также средств вычислительной техники создавались СУБД, поддерживающие различные даталогические модели.

Сначала стали использовать иерархические даталогические модели. Простота организации, наличие заранее заданных связей между сущностями, сходство с физическими моделями данных позволяли добиваться приемлемой производительности иерархических СУБД на медленных ЭВМ с весьма ограниченными объемами памяти. Но, если данные не имели древовидной структуры, то возникала масса сложностей при построении иерархической модели и желании добиться нужной производительности.

Сетевые модели также создавались для мало ресурсных ЭВМ. Это достаточно сложные структуры, состоящие из «наборов» – поименованных двухуровневых деревьев. «Наборы» соединяются с помощью «записей-связок», образуя цепочки и т.д. При разработке сетевых моделей было выдумано множество «маленьких хитростей», позволяющих увеличить про-

изводительность СУБД, но существенно усложнивших последние. Прикладной программист должен знать массу терминов, изучить несколько внутренних языков СУБД, детально представлять логическую структуру базы данных для осуществления навигации среди различных экземпляров, наборов, записей и т.п. Один из разработчиков операционной системы UNIX сказал: «Сетевая база – это самый верный способ потерять данные».

Сложность практического использования иерархических и сетевых СУБД заставляла искать иные способы представления данных. В конце 60-х годов появились СУБД на основе инвертированных файлов, отличающиеся простотой организации и наличием весьма удобных языков манипулирования данными. Однако такие СУБД обладают рядом ограничений на количество файлов для хранения данных, количество связей между ними, длину записи и количество ее полей.

Сегодня наиболее распространена реляционная модель, которая будет подробно рассмотрена в данном курсе.

Физическая организация данных оказывает основное влияние на эксплуатационные характеристики БД. Разработчики СУБД пытаются создать наиболее производительные физические модели данных, предлагая пользователям тот или иной инструментарий для настройки модели под конкретную БД. Разнообразие способов корректировки физических моделей современных промышленных СУБД не позволяет рассмотреть их в этом разделе.

Трехуровневая архитектура (инфологический, даталогический и физический уровни) позволяет обеспечить независимость хранимых данных от использующих их программ. АБД может при необходимости переписать хранимые данные на другие носители информации и (или) реорганизовать их физическую структуру, изменив лишь физическую модель данных. АБД может подключить к системе любое число новых пользователей (новых приложений), дополнив, если надо, даталогическую модель. Указанные изменения физической и даталогической моделей не будут замечены существующими пользователями системы (окажутся «прозрачными» для них), так же как не будут замечены и новые пользователи. Следовательно, независимость данных обеспечивает возможность развития системы баз данных без разрушения существующих приложений.

1 БАНКИ ДАННЫХ

1.1 Пользователи и персонал банка данных

Как любой программно-организационно-технический комплекс, банк данных существует во времени и в пространстве. Он имеет определенные стадии своего развития:

- Проектирование.
- Реализация.
- Эксплуатация.
- Модернизация и развитие.
- Полная реорганизация.

На каждом этапе своего существования с банком данных связаны разные категории пользователей. Определим основные категории пользователей и их роль в функционировании банка данных.

Конечные пользователи. Это основная категория пользователей, в интересах которых и создается банк данных. В зависимости от особенностей создаваемого банка данных круг его конечных пользователей может существенно различаться. Это могут быть случайные пользователи, обращающиеся к БД время от времени за получением некоторой информации, а могут быть регулярные пользователи. В качестве случайных пользователей могут рассматриваться, например, возможные клиенты вашей фирмы, просматривающие каталог вашей продукции или услуг с обобщенным или подробным описанием того и другого. Регулярными пользователями могут быть ваши сотрудники, работающие со специально разработанными для них программами, которые обеспечивают автоматизацию их деятельности при выполнении своих должностных обязанностей. Например, менеджер, планирующий работу сервисного отдела компьютерной фирмы, имеет в своем распоряжении программу, которая помогает ему планировать и распределять текущие заказы, контролировать ход их выполнения, заказывать на складе необходимые комплектующие для новых заказов. Главный принцип состоит в том, что от конечных пользователей не должно требоваться каких-либо специальных знаний в области вычислительной техники и языковых средств.

Администраторы банка данных. Это группа персонала, которая на начальной стадии разработки банка данных отвечает за его оптимальную организацию с точки зрения одновременной работы множества конечных пользователей, на стадии эксплуатации отвечает за корректность работы данного банка информации в многопользовательском режиме. На стадии развития и реорганизации эта группа отвечает за возможность корректной реорганизации банка без изменения или прекращения его текущей эксплуатации.

Разработчики и администраторы приложений. Это группа персонала, которая функционирует во время проектирования, создания и реорганизации банка данных. Администраторы приложений координируют работу разработчиков при разработке конкретного приложения или группы приложений, объединенных в функциональную подсистему. Разработчики конкретных приложений работают с той частью информации из базы данных, которая требуется для конкретного приложения.

Не в каждом банке данных могут быть выделены все эти группы. Например, при разработке информационных систем с использованием настольных СУБД администратор банка данных, администратор приложений и разработчик часто существовали в одном лице. Однако при построении современных сложных корпоративных баз данных, которые используются для автоматизации всех или большей части бизнес-процессов в крупной фирме или корпорации, могут существовать и группы администраторов приложений, и отделы разработчиков. Наиболее сложные обязанности возложены на группу администратора БД (АБД). Рассмотрим их более подробно.

В составе группы администратора БД должны быть:

- системные аналитики;
- проектировщики структур данных и внешнего по отношению к банку данных информационного обеспечения;
- проектировщики технологических процессов обработки данных;
- системные и прикладные программисты;
- операторы и специалисты по техническому обслуживанию.

Если речь идет о коммерческом банке данных, то важную роль здесь играют специалисты по маркетингу.

1.2 Основные функции группы администратора БД

1. Анализ предметной области:

- описание предметной области;
- выявление ограничений целостности;
- определение статуса (доступности, секретности) информации;
- определение потребностей пользователей;
- определение соответствия «данные – пользователь»;
- определение объемно-временных характеристик обработки данных.
- 2. Проектирование структуры БД:
- определение состава и структуры файлов БД и связей между ними;
- выбор методов упорядочения данных и методов доступа к информации;
- описание БД на языке описания данных (ЯОД).

3. Задание ограничений целостности при описании структуры БД и процедур обработки БД:

- задание декларативных ограничений целостности, присущих предметной области;
- определение динамических ограничений целостности, присущих предметной области в процессе изменения информации, хранящейся в БД;
- определение ограничений целостности, вызванных структурой БД;
- разработка процедур обеспечения целостности БД при вводе и корректировке данных;
- определение ограничений целостности при параллельной работе пользователей в многопользовательском режиме.

4. Первоначальная загрузка и ведение БД:

- разработка технологии первоначальной загрузки БД, которая будет отличаться от процедуры модификации и дополнения данными при штатном использовании базы данных;
- разработка технологии проверки соответствия введенных данных реальному состоянию предметной области. База данных моделирует реальные объекты некоторой предметной области и взаимосвязи между ними, и на момент начала штатной эксплуатации эта модель должна полностью соответствовать состоянию объектов предметной области на данный момент времени;
- в соответствии с разработанной технологией первоначальной загрузки может понадобиться проектирование системы первоначального ввода данных.

5. Защита данных:

- определение системы паролей, принципов регистрации пользователей, создание групп пользователей, обладающих одинаковыми правами доступа к данным;
- разработка принципов защиты конкретных данных и объектов проектирования; разработка специализированных методов кодирования информации при ее циркуляции в локальной и глобальной информационных сетях;
- разработка средств фиксации доступа к данным и попыток нарушения системы защиты;
- тестирование системы защиты;
- исследование случаев нарушения системы защиты и развитие динамических методов защиты информации в БД.

6. Обеспечение восстановления БД:

- разработка организационных средств архивирования и принципов восстановления БД;
- разработка дополнительных программных средств и технологических процессов восстановления БД после сбоев.

7. Анализ обращений пользователей БД: сбор статистики по характеру запросов, по времени их выполнения, по требуемым выходным документам.

8. Анализ эффективности функционирования БД:

- анализ показателей функционирования БД;
- планирование реструктуризации (изменение структуры) БД и реорганизации БнД.

9. Работа с конечными пользователями:

- сбор информации об изменении предметной области;
- сбор информации об оценке работы БД;
- обучение пользователей, консультирование пользователей;
- разработка необходимой методической и учебной документации по работе конечных пользователей.

10. Подготовка и поддержание системных средств:

- анализ существующих на рынке программных средств и анализ возможности и необходимости их использования в рамках БД;
- разработка требуемых организационных и программно-технических мероприятий по развитию БД;
- проверка работоспособности закупаемых программных средств перед подключением их к БД;
- курирование подключения новых программных средств к БД.

11. Организационно-методическая работа по проектированию БД:

- выбор или создание методики проектирования БД;
- определение целей и направления развития системы в целом;
- планирование этапов развития БД;
- разработка общих словарей-справочников проекта БД и концептуальной
- модели;
- стыковка внешних моделей разрабатываемых приложений;
- курирование подключения нового приложения к действующей БД;
- обеспечение возможности комплексной отладки множества приложений, взаимодействующих с одной БД.

2 ИНФОЛОГИЧЕСКАЯ МОДЕЛЬ ДАННЫХ

2.1 Семантическое моделирование данных

Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к упоминавшейся нами проблеме представления ограничений целостности. Процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области («сущностей») и выявления связей между этими сущностями. Семантика предметной области представляется в виде инфологической модели, поэтому построение инфологической модели называют семантическим моделированием предметной области.

Потребности проектировщиков баз данных в удобных и мощных средствах моделирования предметной области вызвали к жизни направление семантических моделей данных. Любая развитая семантическая модель данных, как и реляционная модель, включает структурную, манипуляционную и целостную части, главным назначением семантических моделей является обеспечение возможности выражения семантики данных.

Прежде чем мы коротко рассмотрим особенности одной из распространенных семантических моделей, остановимся на их возможных применениях.

Наиболее часто на практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования.

Менее часто реализуется автоматизированная компиляция концептуальной схемы в реляционную. При этом известны два подхода: на основе явного представления концептуальной схемы как исходной информации для компилятора и построения интегрированных систем проектирования с автоматизированным созданием концептуальной схемы на основе интервью с экспертами предметной области. И в том, и в другом случае в результате производится реляционная схема базы данных в третьей нормальной форме (более точно следовало бы сказать, что автору неизвестны системы, обеспечивающие более высокий уровень нормализации).

Наконец, третья возможность, которая еще не вышла (или только выходит) за пределы исследовательских и экспериментальных проектов, – это работа с базой данных в семантической модели, т.е. СУБД, основанные на семантических моделях данных. При этом снова рассматриваются два варианта: обеспечение пользовательского интерфейса на основе семанти-

ческой модели данных с автоматическим отображением конструкций в реляционную модель данных (это задача примерно такого же уровня сложности, как автоматическая компиляция концептуальной схемы базы данных в реляционную схему) и прямая реализация СУБД, основанная на какой-либо семантической модели данных. Наиболее близко ко второму подходу находятся современные *объектно-ориентированные СУБД*, модели данных которых по многим параметрам близки к семантическим моделям.

2.2 Основные понятия инфологической модели

Цель инфологического моделирования – обеспечение наиболее естественных для человека способов сбора и представления той информации, которую предполагается хранить в создаваемой базе данных. Поэтому инфологическую модель данных пытаются строить по аналогии с естественным языком (последний не может быть использован в чистом виде из-за сложности компьютерной обработки текстов и неоднозначности любого естественного языка). Основными конструктивными элементами инфологических моделей являются сущности, связи между ними и их свойства (атрибуты).

Сущность – любой различимый объект (объект, который мы можем отличить от другого), информацию о котором необходимо хранить в базе данных. Сущностями могут быть люди, места, самолеты, рейсы, вкус, цвет и т.д. Необходимо различать такие понятия, как тип сущности и экземпляр сущности. Понятие тип сущности относится к набору однородных личностей, предметов, событий или идей, выступающих как целое. *Экземпляр* сущности относится к конкретной вещи в наборе. Например, типом сущности может быть ГОРОД, а экземпляром – Москва, Киев и т.д.

Атрибут – поименованная характеристика сущности. Его наименование должно быть уникальным для конкретного типа сущности, но может быть одинаковым для различного типа сущностей (например, ЦВЕТ может быть определен для многих сущностей: СОБАКА, АВТОМОБИЛЬ, ДЫМ и т.д.). Атрибуты используются для определения того, какая информация должна быть собрана о сущности. Примерами атрибутов для сущности АВТОМОБИЛЬ являются ТИП, МАРКА, НОМЕРНОЙ ЗНАК, ЦВЕТ и т.д. Здесь также существует различие между типом и экземпляром. Тип атрибута ЦВЕТ имеет много экземпляров или значений:

Красный, Синий, Банановый, Белая ночь и т.д.,

однако каждому экземпляру сущности присваивается только одно значение атрибута.

Абсолютное различие между типами сущностей и атрибутами отсутствует. Атрибут является таковым только в связи с типом сущности. В дру-

гом контексте атрибут может выступать как самостоятельная сущность. Например, для автомобильного завода цвет – это только атрибут продукта производства, а для лакокрасочной фабрики цвет – тип сущности.

Ключ – минимальный набор атрибутов, по значениям которых можно однозначно найти требуемый экземпляр сущности. Минимальность означает, что исключение из набора любого атрибута не позволяет идентифицировать сущность по оставшимся. Для сущности Расписание ключом является атрибут Номер_рейса или набор: Пункт_отправления, Время_вылета и Пункт_назначения (при условии, что из пункта в пункт вылетает в каждый момент времени один самолет).

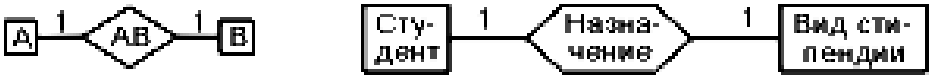
Связь – ассоциирование двух или более сущностей. Если бы назначением базы данных было только хранение отдельных, не связанных между собой данных, то ее структура могла бы быть очень простой. Однако одно из основных требований к организации базы данных – это обеспечение возможности отыскания одних сущностей по значениям других, для чего необходимо установить между ними определенные связи. А так как в реальных базах данных нередко содержатся сотни или даже тысячи сущностей, то теоретически между ними может быть установлено более миллиона связей. Наличие такого множества связей и определяет сложность инфологических моделей.

2.3 Характеристика связей и язык моделирования

При построении инфологических моделей можно использовать язык ER-диаграмм (от англ. Entity-Relationship, т.е. сущность-связь). В них сущности изображаются помеченными прямоугольниками, ассоциации – помеченными ромбами или шестиугольниками, атрибуты – помеченными овалами, а связи между ними – ненаправленными ребрами, над которыми может проставляться степень связи (1 или буква, заменяющая слово «много») и необходимое пояснение.

Между двумя сущностям, например, А и В возможны четыре вида связей.

Первый тип – связь ОДИН-К-ОДНОМУ (1:1): в каждый момент времени каждому представителю (экземпляру) сущности А соответствует 1 или 0 представителей сущности В:



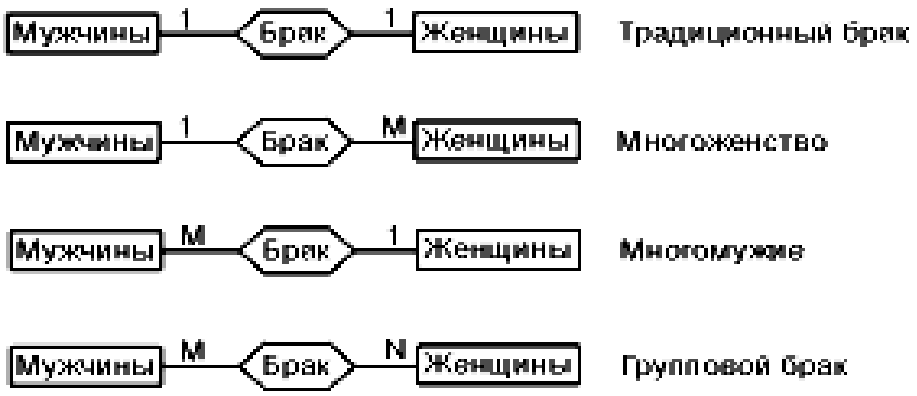
Студент может не «заработать» стипендию, получить обычную или одну из повышенных стипендий.

Второй тип – связь ОДИН-КО-МНОГИМ (1:M): одному представителю сущности A соответствуют 0, 1 или несколько представителей сущности B.

Квартира может пустовать, в ней может жить один или несколько жильцов.

Так как между двумя сущностями возможны связи в обоих направлениях, то существует еще два типа связи МНОГИЕ-К-ОДНОМУ (M:1) и МНОГИЕ-КО-МНОГИМ (M:N).

Например, если связь между сущностями МУЖЧИНЫ и ЖЕНЩИНЫ называется БРАК, то существует четыре возможных варианта такой связи:

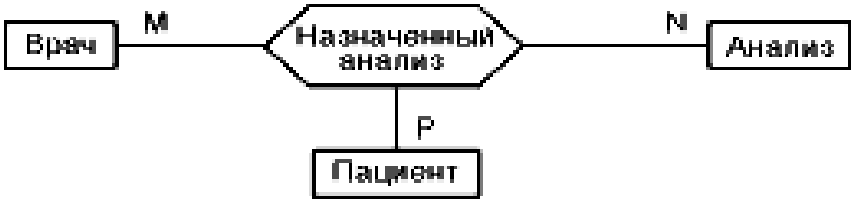


Характер связей между сущностями не ограничивается перечисленными. Существуют и более сложные связи.

Множественные связи между одними и теми же сущностями

Пациент, имея одного лечащего врача, может иметь также несколько врачей-консультантов; врач может быть лечащим врачом нескольких пациентов и может одновременно консультировать несколько других пациентов.

Тренарные связи



Врач может назначить несколько пациентов на несколько анализов, анализ может быть назначен несколькими врачами нескольким пациентам и пациент может быть назначен на несколько анализов несколькими врачами.

Возможны связи более высоких порядков, семантика которых иногда очень сложна.

В приведенных примерах для повышения иллюстративности рассматриваемых связей не показаны атрибуты сущностей и ассоциаций во всех ER-диаграммах. Так, ввод лишь нескольких основных атрибутов в описание брачных связей значительно усложнит ER-диаграмму (рис. 2.1а). В связи с этим язык ER-диаграмм используется для построении небольших моделей и иллюстрации отдельных фрагментов больших.

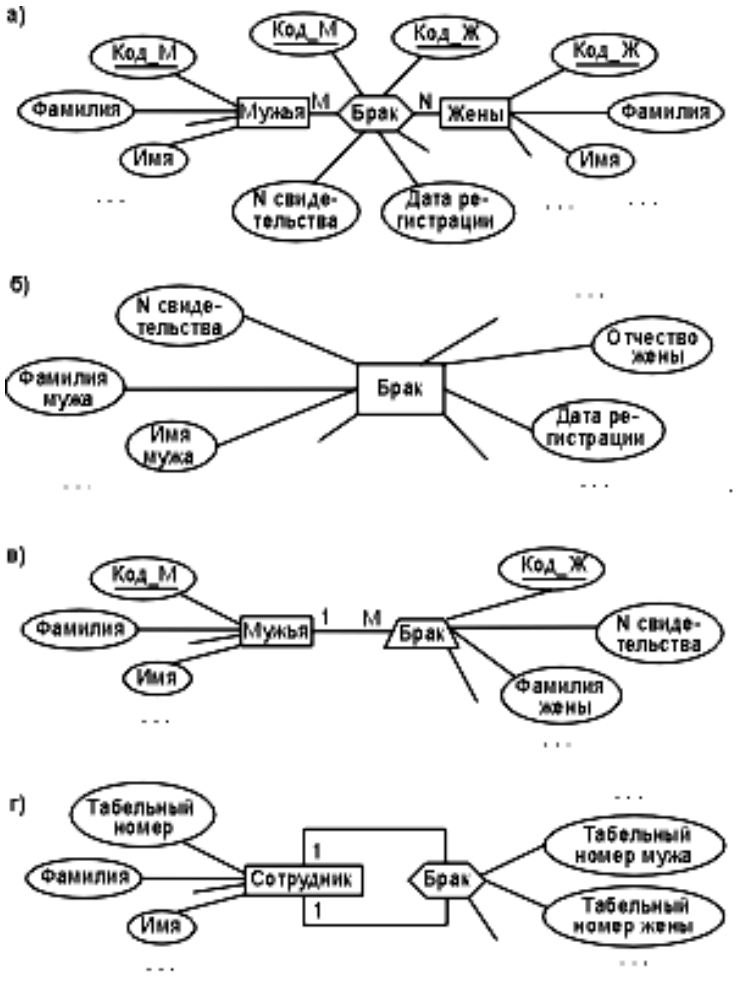


Рис. 2.1. Примеры ER-диаграмм

Чаще же применяется менее наглядный, но более содержательный язык инфологического моделирования (ЯИМ), в котором сущности и ассоциации представляются предложениями вида:

СУЩНОСТЬ (атрибут 1, атрибут 2 , ..., атрибут *n*)

АССОЦИАЦИЯ [СУЩНОСТЬ *S*1, СУЩНОСТЬ *S*2, ...] (атрибут 1, атрибут 2, ..., атрибут *n*),

где *S* – степень связи, а атрибуты, входящие в ключ, должны быть отмечены с помощью подчеркивания.

Так, рассмотренный выше пример множества связей между сущностями может быть описан на ЯИМ следующим образом:

Врач (Номер_врача, Фамилия, Имя, Отчество, Специальность)
Пациент (Регистрационный_номер, Номер койки, Фамилия,
Имя, Отчество, Адрес, Дата рождения, Пол)
Лечащий_врач [Врач 1, Пациент M]
(Номер_врача, Регистрационный_номер)
Консультант [Врач M, Пациент N]
(Номер_врача, Регистрационный_номер).

Для выявления связей между сущностями необходимо, как минимум, определить сами сущности. Но это не простая задача, так как в разных предметных областях один и тот же объект может быть сущностью, атрибутом или ассоциацией. Проиллюстрируем такое утверждение на примерах, связанных с описанием брачных связей.

Отдел записей актов гражданского состояния (ЗАГС) имеет дело не со всеми людьми, а только с теми, кто обратился с просьбой о регистрации брака, рождения или смерти. Поэтому в странах, где допускаются лишь традиционные браки, отделы ЗАГС могут размещать сведения о регистрируемых браках в единственной сущности:

Брак (Номер_свидетельства, Фамилия_мужа, Имя_мужа,
Отчество_мужа, Дата_рождения_мужа, Фамилия_жены,
... , Дата_регистрации, Место_регистрации, ...),

ER-диаграмма которой приведена на рисунке 2.1б.

Теперь рассмотрим ситуацию, когда отдел ЗАГС расположен в стране, допускающей многоженство. Если для регистрации браков использовать сущность «Брак», то будут дублироваться сведения о мужьях, имеющих несколько жен, как показано в таблице 2.1.

Таблица 2.1

Номер свидетельства	Фамилия мужа	Фамилия жены	Дата регистрации, г.
1-ЮБ-154745	Петухов ...	Курочкина ...	06/03/1991
1-ЮБ-163489	Петухов ...	Пеструшкина ...	11/08/1991
1-ЮБ-169887	Петухов ...	Рябова ...	12/12/1992
1-ЮБ-169878	Селезнев ...	Уточкина ...	12/12/1992
1-ЮБ-154746	Парасюк ...	Свинюшкина ...	06/03/1991
1-ЮБ-169879	Парасюк ...	Хаврония ...	12/12/1992

Дублирование можно исключить созданием дополнительной сущности «Мужья»

Мужья (Код_М, Фамилия, Имя, Отчество, Дата рождения, Место рождения)

и заменой сущности «Брак» характеристикой со ссылкой на соответствующее описание в сущности «Мужья».

Брак (Номер свидетельства, Код_М, Фамилия жены, ...,
Дата регистрации, ...){Мужья}.

ER-диаграмма связи этих сущностей показана на рисунке 2.1в, а пример их экземпляров в таблицах 2.2 и 2.3.

Таблица 2.2

Код_М	Фамилия Имя Отчество	Год рождения	Место рождения
111	Петухов Альфред Остапович	1971	г. Цапелька
112	Селезнев Вавила Абрамович	1973	г. Гусев
113	Парасюк Гораций Федулович	1972	г. Свиньин

Таблица 2.3

Номер свидетельства	Код_М	Фамилия жены	Имя жены	Дата регистрации
1-ЮБ 154745	111	Курочкина	Августина	06/03/1991
1-ЮБ 163489	111	Пеструшкина	Мариана	11/08/1991
1-ЮБ 169877	111	Рябова	Милана	12/12/1992
1-ЮБ 169878	112	Уточкина	Вероника	12/12/1992 ...
1-ЮБ 154746	113	Свинюшкина	Эльвира	06/03/1991
1 ЮБ 169879	113	Хаврония	Руфина	12/12/1992

Наконец, рассмотрим случай, когда какой-либо организации потребовались данные о наличии в ней семейных пар, а для хранения сведений о сотрудниках уже имеется сущность

Сотрудники (Табельный_номер, Фамилия, Имя, ...).

Использование рассмотренной сущности «Брак» нецелесообразно, т.к. в «Сотрудники» уже есть фамилии, имена и отчества супругов. Поэтому создадим ассоциацию

Брак [Сотрудник 1, Сотрудник 1]
(Табельный_номер_мужа, Табельный_номер_жены, ...),

связывающую между собой определенные экземпляры сущности «Сотрудники» (рис. 2.1г).

Что же такое «связь»? В ER-диаграммах это линия, соединяющая геометрические фигуры, изображающие сущности, атрибуты, ассоциации и другие информационные объекты. В тексте же этот термин используется для указания на взаимозависимость сущностей. Если эта взаимозависимость имеет атрибуты, то она называется ассоциацией.

2.4 Классификация сущностей

Настал момент разобраться в терминологии. К. Дейт определяет три основных класса сущностей: стержневые, ассоциативные и характеристические, а также подкласс ассоциативных сущностей – обозначения.

Стержневая сущность (стержень) – это независимая сущность (несколько подробнее она будет определена ниже).

Ассоциативная сущность (ассоциация) – это связь вида «многие-ко-многим» («-ко-многим» и т.д.) между двумя или более сущностями или экземплярами сущности (как в последнем примере). Ассоциации рассматриваются как полноправные сущности:

- они могут участвовать в других ассоциациях и обозначениях точно так же, как стержневые сущности;
- могут обладать свойствами, т.е. иметь не только набор ключевых атрибутов, необходимых для указания связей, но и любое число других атрибутов, характеризующих связь.

Например, ассоциации «Брак» содержат ключевые атрибуты «Код_М», «Код_Ж» и «Табельный номер мужа», «Табельный номер жены», а также уточняющие атрибуты «Номер свидетельства», «Дата регистрации», «Место_регистрации», «Номер записи в книгу ЗАГС» и т.д.

Характеристическая сущность (характеристика) – это связь вида «многие-к-одной» или «одна-к-одной» между двумя сущностями (частный случай ассоциации). Единственная цель характеристики в рамках рассматриваемой предметной области состоит в описании или уточнении некоторой другой сущности. Необходимость в них возникает в связи с тем, что сущности реального мира имеют иногда многозначные свойства. Муж может иметь несколько жен, книга – несколько характеристик переиздания (исправленное, дополненное, переработанное) и т.д.

Существование характеристики полностью зависит от характеризуемой сущности: женщины лишаются статуса жен, если умирает их муж.

Для описания характеристики используется новое предложение ЯИМ, имеющее в общем случае вид:

ХАРАКТЕРИСТИКА (атрибут 1, атрибут 2, ...)
{СПИСОК ХАРАКТЕРИЗУЕМЫХ СУЩНОСТЕЙ}.

Расширим также язык ER-диаграмм, введя для изображения характеристики трапецию (рис. 2.2).

Рис. 2.2. Элементы расширенного языка ER-диаграмм

Обозначающая сущность или обозначение – это связь вида «многие-к-одной» или «одна-к-одной» между двумя сущностями и отличается от характеристики тем, что не зависит от обозначаемой сущности.

Рассмотрим пример, связанный с зачислением сотрудников в различные отделы организации.

При отсутствии жестких правил (сотрудник может одновременно зачисляться в несколько отделов или не зачисляться ни в один отдел) необходимо создать описание с ассоциацией Зачисление:

Отделы (Номер отдела, Название отдела, ...)
Служащие (Табельный номер, Фамилия, ...)
Зачисление [Отделы M, Служащие N] (Номер отдела, Табельный номер, Дата зачисления).

Однако при условии, что каждый из сотрудников должен быть обязательно зачислен в один из отделов, можно создать описание с обозначением Служащие:

Отделы (Номер отдела, Название отдела, ...)
Служащие (Табельный номер, Фамилия, ... , Номер отдела, Дата зачисления)[Отделы]

В данном примере служащие имеют независимое существование (если удаляется отдел, то из этого не следует, что также должны быть удалены служащие такого отдела). Поэтому они не могут быть характеристиками отделов и названы обозначениями.

Обозначения используют для хранения повторяющихся значений больших текстовых атрибутов: «кодификаторы» изучаемых студентами дисциплин, наименований организаций и их отделов, перечней товаров и т.п.

Описание обозначения внешне отличается от описания характеристики только тем, что обозначаемые сущности заключается не в фигурные скобки, а в квадратные:

ОБОЗНАЧЕНИЕ (атрибут 1, атрибут 2, ...)[СПИСОК ОБОЗНАЧАЕМЫХ СУЩНОСТЕЙ].

Как правило, обозначения не рассматриваются как полноправные сущности, хотя это не привело бы к какой-либо ошибке.

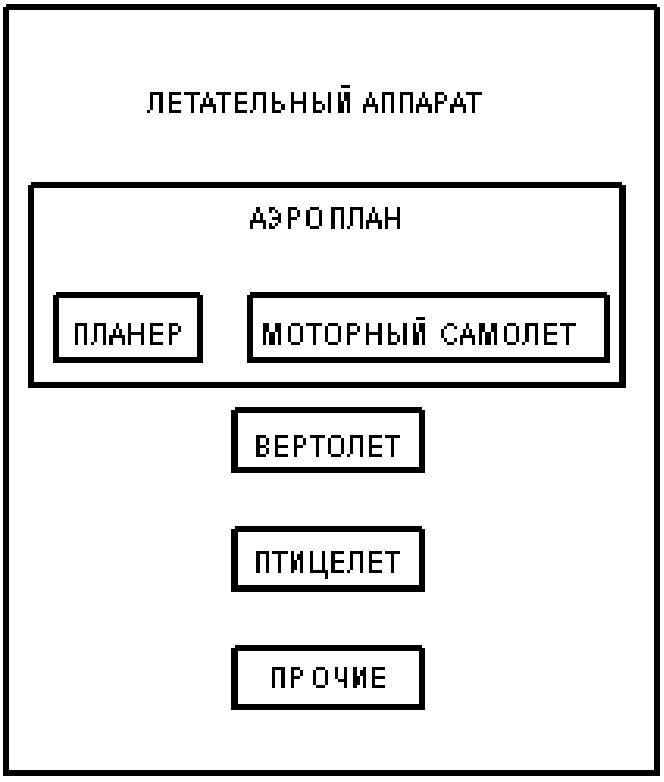
Обозначения и характеристики не являются полностью независимыми сущностями, поскольку они предполагают наличие некоторой другой сущности, которая будет «обозначаться» или «характеризоваться». Однако они все же представляют собой частные случаи сущности и могут, конечно, иметь свойства, могут участвовать в ассоциациях, обозначениях и иметь свои собственные (более низкого уровня) характеристики. Подчеркнем также, что все экземпляры характеристики должны быть обязательно связаны с каким-либо экземпляром характеризуемой сущности. Однако допускается, чтобы некоторые экземпляры характеризуемой сущности не имели связей. Правда, если это касается браков, то сущность «Мужья» должна быть заменена на сущность «Мужчины» (нет мужа без жены).

2.4.1 Супертипы и подтипы

Сущность может быть расщеплена на два или более взаимно исключающих подтипа, каждый из которых включает общие атрибуты и/или связи. Эти общие атрибуты и/или связи явно определяются один раз на более высоком уровне. В подтипах могут определяться собственные атрибуты и/или связи. В принципе подтипизация может продолжаться на более низких уровнях, но опыт показывает, что в большинстве случаев оказывается достаточно двух-трех уровней.

Сущность, на основе которой определяются подтипы, называется *супертипом*. *Подтипы* должны образовывать полное множество, т.е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для полноты приходится определять дополнительный подтип ПРОЧИЕ.

Пример: Супертип ЛЕТАТЕЛЬНЫЙ АППАРАТ



Как полагается это читать? От супертипа: ЛЕТАТЕЛЬНЫЙ АППАРАТ, который должен быть АЭРОПЛАНОМ, ВЕРТОЛЕТОМ, ПТИЦЕЛЕТОМ или ДРУГИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ. От подтипа: ВЕРТОЛЕТ, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА. От подтипа, который является одновременно супертипом: АЭРОПЛАН, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА и должен быть ПЛАНЕРОМ или МОТОРНЫМ САМОЛЕТОМ.

Иногда удобно иметь два или более разных разбиения сущности на подтипы. Например, сущность ЧЕЛОВЕК может быть разбита на подтипы по профессиональному признаку (ПРОГРАММИСТ, ДОЯРКА и т.д.), а может – по половому признаку (МУЖЧИНА, ЖЕНЩИНА).

2.5 Первичные и внешние ключи

Напомним, что ключ или возможный ключ – это минимальный набор атрибутов, по значениям которых можно однозначно найти требуемый экземпляр сущности. Минимальность означает, что исключение из набора любого атрибута не позволяет идентифицировать сущность по оставшимся. Каждая сущность обладает хотя бы одним возможным ключом. Один из них принимается за первичный ключ. При выборе первичного ключа следует отдавать предпочтение несоставным ключам или ключам, составленным из минимального числа атрибутов. Нецелесообразно также ис-

пользовать ключи с длинными текстовыми значениями (предпочтительнее использовать целочисленные атрибуты). Так, для идентификации студента можно использовать либо уникальный номер зачетной книжки, либо набор из фамилии, имени, отчества, номера группы и, может быть, дополнительных атрибутов, так как не исключено появление в группе двух студентов с одинаковыми фамилиями, именами и отчествами. Плохо также использовать в качестве ключа не номер блюда, а его название, например « Закуска из плавленых сырков «Дружба» с ветчиной и соленым огурцом» или «Заяц в сметане с картофельными крокетами и салатом из красной капусты».

Не допускается, чтобы первичный ключ стержневой сущности (любой атрибут, участвующий в первичном ключе) принимал неопределенное значение. Иначе возникнет противоречивая ситуация: появится не обладающий индивидуальностью, и не существующий экземпляр стержневой сущности. По тем же причинам необходимо обеспечить уникальность первичного ключа.

2.5.1 Внешние ключи

Если сущность С связывает сущности А и В, то она должна включать внешние ключи, соответствующие первичным ключам сущностей А и В.

Если сущность В обозначает сущность А, то она должна включать внешний ключ, соответствующий первичному ключу сущности А. В п. 2.4 рассматривался пример, где «Служащие» обозначали «Отделы» и включали внешний ключ «Номер отдела», соответствующий первичному ключу сущности «Отделы».

Переопределим теперь стержневую сущность как сущность, которая не имеет внешних ключей, т.е. не является ни ассоциацией, ни обозначением, ни характеристикой.

Связь между первичными и внешними ключами сущностей иллюстрируется на рисунке 2.3.

Здесь для обозначения любой из ассоциируемых сущностей (стержней, характеристик, обозначений или даже ассоциаций) используется новый обобщающий термин «Цель» или «Целевая сущность».

Таким образом, при рассмотрении проблемы выбора способа представления ассоциаций и обозначений в базе данных основной вопрос, на который следует получить ответ: «Каковы внешние ключи?» И далее, для каждого внешнего ключа необходимо решить три вопроса.

1) Может ли данный внешний ключ принимать неопределенные значения (NULL-значения)? Иначе говоря, может ли существовать некоторый экземпляр сущности данного типа, для которого неизвестна целевая сущность, указываемая внешним ключом? В случае поставок это, вероятно, невозможно – поставка, осуществляемая неизвестным поставщиком, или поставка неизвестного продукта не имеют смысла. Но в случае с сотрудниками такая ситуация однако могла бы иметь смысл – вполне возможно,

что какой-либо сотрудник в данный момент не зачислен вообще ни в какой отдел. Заметим, что ответ на данный вопрос не зависит от прихоти проектировщика базы данных, а определяется фактическим образом действий, принятым в той части реального мира, которая должна быть представлена в рассматриваемой базе данных. Подобные замечания имеют отношение и к вопросам, обсуждаемым ниже.

2) Что должно случиться при попытке УДАЛЕНИЯ целевой сущности, на которую ссылается внешний ключ? Например, при удалении поставщика, который осуществил, по крайней мере, одну поставку.

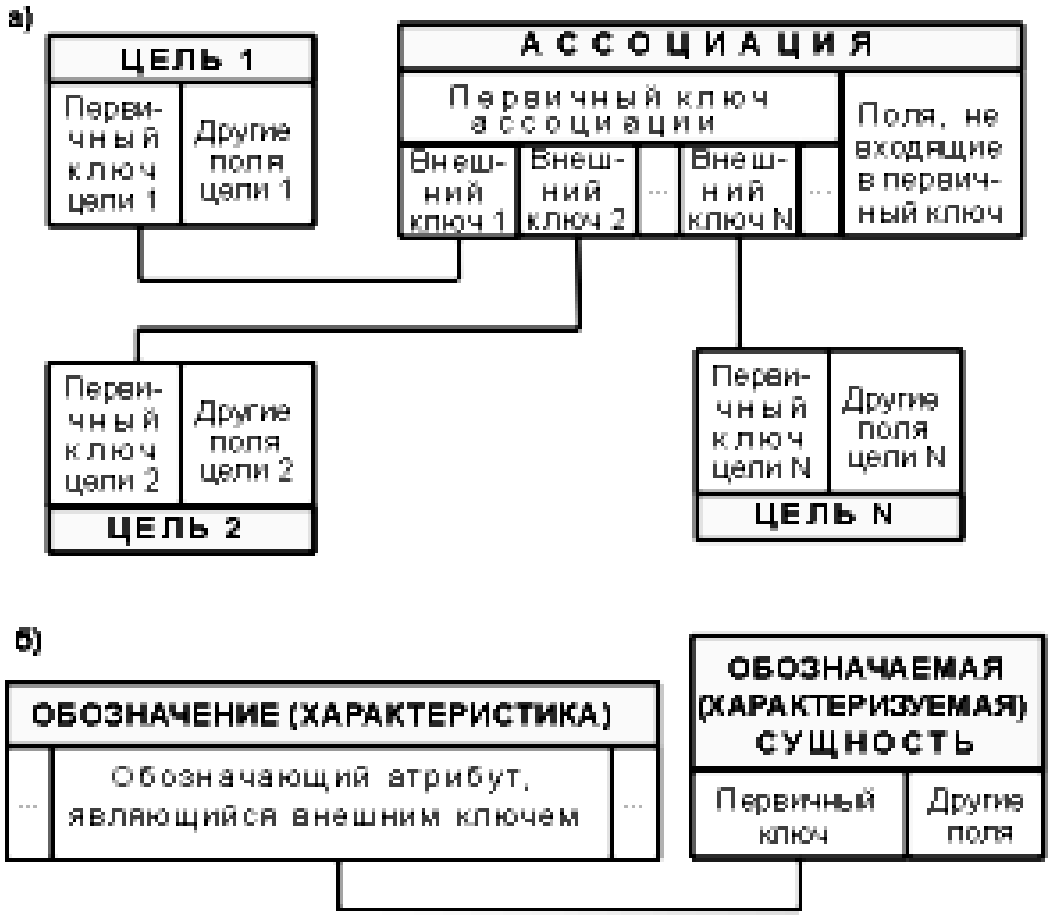


Рис. 2.3. Структуры сущностей:
а – ассоциации; б – обозначения (характеристики)

Существует три возможности.

КАСКАДИРОВАНИЕ – операция удаления «каскадируется» с тем, чтобы удалить также поставки этого поставщика.

ОГРАНИЧЕНИЕ – удаляются лишь те поставщики, которые еще не осуществляли поставок. Иначе операция удаления отвергается.

УСТАНОВКА – Для всех поставок удаляемого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а за-

тем этот поставщик удаляется. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

3) Что должно происходить при попытке ОБНОВЛЕНИЯ первичного ключа целевой сущности, на которую ссылается некоторый внешний ключ? Например, может быть предпринята попытка обновить номер такого поставщика, для которого имеется, по крайней мере, одна соответствующая поставка. Этот случай для определенности снова рассмотрим подробнее. Имеются те же три возможности, как и при удалении.

КАСКАДИРОВАНИЕ – операция обновления «каскадируется» с тем, чтобы обновить также и внешний ключ в поставках этого поставщика.

ОГРАНИЧЕНИЕ – обновляются первичные ключи лишь тех поставщиков, которые еще не осуществляли поставок. Иначе операция обновления отвергается.

УСТАНОВКА – для всех поставок такого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а затем обновляется первичный ключ поставщика. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

Таким образом, для каждого внешнего ключа проектировщик базы данных должен специфицировать не только поле или комбинацию полей, составляющих этот внешний ключ, и целевую сущность, которая идентифицируется этим ключом, но также и ответы на указанные выше вопросы (три ограничения, которые относятся к этому внешнему ключу).

Наконец, о характеристиках – обозначающих сущностях, существование которых зависит от типа обозначаемых сущностей. Обозначение представляется внешним ключом в таблице, соответствующей этой характеристике. Но три рассмотренных выше ограничения на внешний ключ для данного случая должны специфицироваться следующим образом:

NULL-значения не допустимы
УДАЛЕНИЕ ИЗ (цель) КАСКАДИРУЕТСЯ
ОБНОВЛЕНИЕ (первичный ключ цели) КАСКАДИРУЕТСЯ

Указанные спецификации представляют зависимость по существованию характеристических сущностей.

3 ДАТАЛОГИЧЕСКИЕ МОДЕЛИ ДАННЫХ

Прежде чем перейти к детальному и последовательному изучению реляционных систем БД, остановимся коротко на ранних СУБД. В этом есть смысл по трем причинам: во-первых, эти системы исторически предшествовали реляционным, и для правильного понимания причин повсеместного перехода к реляционным системам нужно знать хотя бы что-нибудь про их предшественников; во-вторых, внутренняя организация реляционных систем во многом основана на использовании методов ранних систем; в-третьих, некоторое знание в области ранних систем будет полезно для понимания путей развития постреляционных СУБД.

Мы ограничиваемся рассмотрением только общих подходов к организации трех типов ранних систем, а именно: систем, основанных на инвертированных списках, иерархических и сетевых систем управления базами данных. Начнем с некоторых наиболее общих характеристик ранних систем.

Эти системы активно использовались в течение многих лет, дольше, чем используется какая-либо из реляционных СУБД. На самом деле некоторые из ранних систем используются даже в наше время, накоплены громадные базы данных, и одной из актуальных проблем информационных систем является использование этих систем совместно с современными системами.

Все ранние системы не основывались на каких-либо абстрактных моделях. Понятие модели данных фактически вошло в обиход специалистов в области БД только вместе с реляционным подходом. Абстрактные представления ранних систем появились позже на основе анализа и выявления общих признаков у различных конкретных систем.

В ранних системах доступ к БД производился на уровне записей. Пользователи этих систем осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный доступ к БД поддерживался только путем создания соответствующих прикладных программ с собственным интерфейсом.

Навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователя самого производить всю оптимизацию доступа к БД, без какой-либо поддержки системы.

3.1 Системы на инвертированных списках

К числу наиболее известных и типичных представителей таких систем относятся Datacom/DB компании Applied Data Research, Inc. (ADR), ориентированная на использование на машинах основного класса фирмы IBM, и Adabas компании Software AG.

Организация доступа к данным на основе инвертированных списков используется практически во всех современных реляционных СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным спискам (индексам). Кстати, когда мы будем рассматривать внутренние интерфейсы реляционных СУБД, вы увидите, что они очень близки к пользовательским интерфейсам систем, основанных на инвертированных списках.

3.1.1 Структуры данных

База данных, организованная с помощью инвертированных списков, похожа на реляционную БД, но с тем отличием, что хранимые таблицы и пути доступа к ним видны пользователям. При этом:

- Строки таблиц упорядочены системой в некоторой физической последовательности.
- Физическая упорядоченность строк всех таблиц может определяться и для всей БД (так делается, например, в Datacom/DB).

Для каждой таблицы можно определить произвольное число ключей поиска, для которых строятся индексы. Эти индексы автоматически поддерживаются системой, но явно видны пользователям.

3.1.2 Манипулирование данными

Поддерживаются два класса операторов:

а) Операторы, устанавливающие адрес записи, среди которых:

- прямые поисковые операторы (например, найти первую запись таблицы по некоторому пути доступа);
- операторы, находящие запись в терминах относительной позиции от предыдущей записи по некоторому пути доступа.

б) Операторы над адресуемыми записями

Типичный набор операторов:

LOCATE FIRST – найти первую запись таблицы T в физическом порядке; возвращает адрес записи;

LOCATE FIRST WITH SEARCH KEY EQUAL – найти первую запись таблицы T с заданным значением ключа поиска K; возвращает адрес записи;

LOCATE NEXT – найти первую запись, следующую за записью с заданным адресом в заданном пути доступа; возвращает адрес записи;

LOCATE NEXT WITH SEARCH KEY EQUAL – найти следующую запись таблицы T в порядке пути поиска с заданным значением K; должно быть соответствие между используемым способом сканирования и ключом K; возвращает адрес записи;

LOCATE FIRST WITH SEARCH KEY GREATER – найти первую запись таблицы T в порядке ключа поиска K со значением ключевого поля, большим заданного значения K; возвращает адрес записи;

RETRIVE – выбрать запись с указанным адресом;

UPDATE – обновить запись с указанным адресом;

DELETE – удалить запись с указанным адресом;

STORE – включить запись в указанную таблицу; операция генерирует адрес записи.

Общие правила определения целостности БД отсутствуют. В некоторых системах поддерживаются ограничения уникальности значений некоторых полей, но в основном все возлагается на прикладную программу.

3.2 Иерархические модели

Типичным представителем иерархической модели данных (наиболее известным и распространенным) является Information Management System (IMS) фирмы IBM. Первая версия появилась в 1968 г. До сих пор поддерживается много баз данных, что создает существенные проблемы с переходом как на новую технологию БД, так и на новую технику.

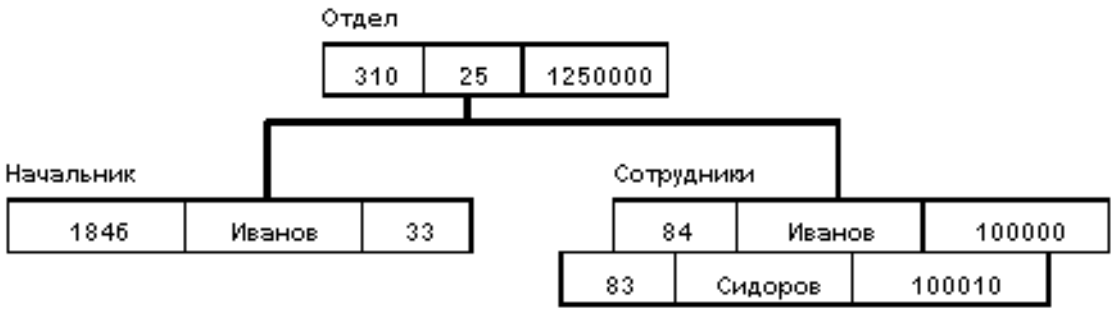
Иерархическая БД состоит из упорядоченного набора деревьев; более точно, из упорядоченного набора нескольких экземпляров одного типа дерева.

Тип дерева состоит из одного «корневого» типа записи и упорядоченного набора из нуля или более типов поддеревьев (каждое из которых является некоторым типом дерева). Тип дерева в целом представляет собой иерархически организованный набор типов записи.

Пример типа дерева (схемы иерархической БД):

Здесь тип записи Отдел является предком для типов Начальник и Сотрудники, а Начальник и Сотрудники – потомки Отдел. Между типами записи поддерживаются связи.

База данных с такой схемой могла бы выглядеть следующим образом (мы показываем один экземпляр дерева):



Все экземпляры данного типа потомка с общим экземпляром типа предка называются *близнецами*. Для БД определен полный порядок обхода – сверху-вниз, слева-направо.

В IMS использовалась оригинальная и нестандартная терминология: «сегмент» вместо «запись», а под «записью БД» понималось все дерево сегментов.

Примерами типичных операторов манипулирования иерархически организованными данными могут быть следующие:

– найти указанное дерево БД (например, отдел 310);
– перейти от одного дерева к другому;
– перейти от одной записи к другой внутри дерева (например, от отдела – к первому сотруднику);
– перейти от одной записи к другой в порядке обхода иерархии;
– вставить новую запись в указанную позицию;
– удалить текущую запись.

Автоматически поддерживается целостность ссылок между предками и потомками. *Основное правило: никакой потомок не может существовать без своего родителя.* Заметим, что аналогичное поддержание целостности по ссылкам между записями, не входящими в одну иерархию, не поддерживается (примером такой «внешней» ссылки может быть содержимое поля Каф_Номер в экземпляре типа записи Куратор).

3.3 Сетевые модели

Типичным представителем является Integrated Database Management System (IDMS) компании Cullinet Software, Inc., предназначенная для использования на машинах основного класса фирмы IBM под управлением большинства операционных систем. Архитектура системы основана на предложениях Data Base Task Group (DBTG) Комитета по языкам программирования Conference on Data Systems Languages (CODASYL), организации, ответственной за определение языка программирования Кобол. Отчет DBTG был опубликован в 1971 г., а в 70-х годах появилось несколько систем, среди которых IDMS.

Сетевой подход к организации данных является расширением иерархического. В иерархических структурах запись-потомок должна иметь в точности одного предка; в сетевой структуре данных потомок может иметь любое число предков.

Сетевая БД состоит из набора записей и набора связей между этими записями, а если говорить более точно, из набора экземпляров каждого типа из заданного в схеме БД набора типов записи и набора экземпляров каждого типа из заданного набора типов связи.

Тип связи определяется для двух типов записи: предка и потомка. Экземпляр типа связи состоит из одного экземпляра типа записи предка и упорядоченного набора экземпляров типа записи потомка. Для данного типа связи L с типом записи предка P и типом записи потомка C должны выполняться следующие два условия:

- каждый экземпляр типа P является предком только в одном экземпляре L;
- каждый экземпляр C является потомком не более, чем в одном экземпляре L.

На формирование типов связи не накладываются особые ограничения; возможны, например, следующие ситуации:

- тип записи потомка в одном типе связи L1 может быть типом записи предка в другом типе связи L2 (как в иерархии).
- данный тип записи P может быть типом записи предка в любом числе типов связи.
- данный тип записи P может быть типом записи потомка в любом числе типов связи.

Может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка. Если L1 и L2 – два типа связи с одним и тем же типом записи предка P и одним и тем же типом записи потомка C, то правила, по которым образуется родство, в разных связях могут различаться.

- Типы записи X и Y могут быть предком и потомком в одной связи и потомком и предком – в другой.
- Предок и потомок могут быть одного типа записи.

Примерный набор операций может быть следующим:

- найти конкретную запись в наборе однотипных записей (инженера Сидорова);
- перейти от предка к первому потомку по некоторой связи (к первому сотруднику отдела 310);

– перейти к следующему потомку в некоторой связи (от Сидорова к Иванову);
– перейти от потомка к предку по некоторой связи (найти отдел Сидорова);
– создать новую запись;
– уничтожить запись;
– модифицировать запись;
– включить в связь;
– исключить из связи;
– переставить в другую связь и т.д.

Простой пример сетевой схемы БД:

3.3.1 Достоинства и недостатки

Сильные места ранних СУБД:
– развитые средства управления данными во внешней памяти на низком уровне;
– возможность построения вручную эффективных прикладных систем;
– возможность экономии памяти за счет разделения подобъектов (в сетевых системах).

Недостатки:
– слишком сложно пользоваться;
– фактически необходимы знания о физической организации;
– прикладные системы зависят от этой организации;
– их логика перегружена деталями организации доступа к БД.

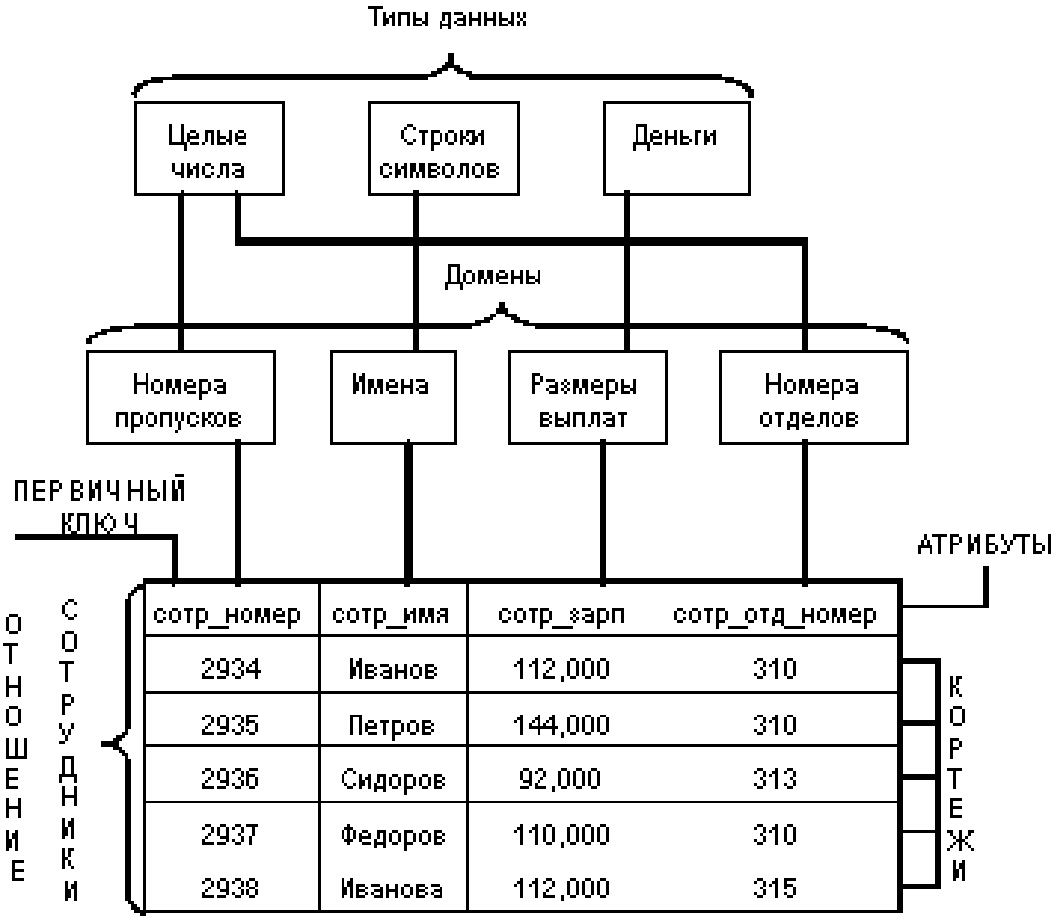
4 РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

Введем на сравнительно неформальном уровне основные понятия реляционных баз данных, а также определим существо реляционной модели данных.

4.1 Базовые понятия

Основными понятиями реляционных баз данных являются тип данных, домен, атрибут, кортеж, первичный ключ и отношение.

Для начала покажем смысл этих понятий на примере отношения СОТРУДНИКИ, содержащего информацию о сотрудниках некоторой организации:



4.1.1 Тип данных

Понятие *тип данных* в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как

«деньги»), а также специальных «темпоральных» данных (дата, время, временной интервал). Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres). В нашем примере мы имеем дело с данными трех типов: строки символов, целые числа и «деньги».

4.1.2 Домен

Понятие *домена* более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат «истина», то элемент данных является элементом домена.

Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа. Например, домен «Имена» в нашем примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака).

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов «Номера пропусков» и «Номера групп» относятся к типу целых чисел, но не являются сравнимыми. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle V.7 оно уже поддерживается.

4.1.3 Схема отношения, схема базы данных

Схема отношения R – это именованное множество пар (имя атрибута, имя домена):

$$S_R = \{(A_i, D_i)\}.$$

Схема отношения представляет в реляционной БД *модель* объекта предметной области. Понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Так, если понятие домена не поддерживается в конкретной реализации СУБД, то вместо него в схеме указывается тип данных для каждого из атрибутов.

Степень отношения – это мощность (количество элементов) схемы отношения. Степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены

на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута).

Схемы двух отношений S_R и S_Q называют *эквивалентными* ($S_R \sim S_Q$), если:

- степени отношений R и Q одинаковы;
- возможна такая перестановка местами атрибутов в схемах, чтобы на одинаковых местах стояли атрибуты с одинаковыми доменами.

Схема БД – это набор именованных схем отношений.

4.1.4 Кортеж, отношение-экземпляр

Кортеж – это множество пар (имя атрибута, значение), которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. Попросту говоря, кортеж – это набор именованных значений атрибутов. Каждый кортеж относится к конкретному объекту предметной области.

«Значение» является допустимым значением домена данного атрибута. Тем самым, степень или «арность» кортежа, т.е. число элементов в нем, совпадает с «арностью» соответствующей схемы отношения.

Отношение-экземпляр – это множество кортежей, соответствующих одной схеме отношения. Иногда схему отношения называют *заголовком отношения*, а отношение-экземпляр – *телом отношения*.

Было бы вполне логично отдельно определять схему отношения, а затем одно или несколько отношений-экземпляров с данной схемой. Однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть *эволюцией схемы базы данных*.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками – кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят «столбец таблицы», имея в виду «атрибут отношения». Когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, мы будем использовать эту житейскую терминологию. Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД.

Реляционная база данных – это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

Как видно, основные структурные понятия реляционной модели данных (если не считать понятия домена) имеют очень простую интуитивную интерпретацию, хотя в теории реляционных БД все они определяются абсолютно формально и точно.

4.2 Фундаментальные свойства отношений

Остановимся теперь на некоторых важных свойствах отношений, которые следуют из приведенных ранее определений.

4.2.1 Отсутствие кортежей-дубликатов

То свойство, что отношения не содержат кортежей-дубликатов, следует из определения отношения как множества кортежей. В классической теории множеств по определению каждое множество состоит из различных элементов.

Из этого свойства вытекает наличие у каждого отношения так называемого *первичного ключа* – набора атрибутов, значения которых однозначно определяют кортеж отношения. Для каждого отношения, по крайней мере, полный набор его атрибутов обладает этим свойством. Однако при формальном определении первичного ключа требуется обеспечение его «минимальности», т.е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства – однозначно определять кортеж. Понятие первичного ключа является исключительно важным в связи с понятием целостности баз данных.

Забегая вперед, заметим, что во многих практических реализациях РСУБД допускается нарушение свойства уникальности кортежей для промежуточных отношений, порождаемых неявно при выполнении запросов. Такие отношения являются не множествами, а мультимножествами, что в ряде случаев позволяет добиться определенных преимуществ, но иногда приводит к серьезным проблемам.

4.2.2 Отсутствие упорядоченности кортежей

Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения-экземпляра как множества кортежей. Отсутствие требования к поддержанию порядка на множестве кортежей отношения дает дополнительную гибкость СУБД при хранении баз данных во внешней памяти и при выполнении запросов к базе данных. Это не противоречит тому, что при формулировании запроса к БД, например, на языке SQL можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых столбцов. Такой результат, вообще говоря, не отношение, а некоторый упорядоченный список кортежей.

4.2.3 Отсутствие упорядоченности атрибутов

Атрибуты отношений не упорядочены, поскольку по определению схема отношения есть множество пар {имя атрибута, имя домена}. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Это свойство теоретически позволяет, например, модифицировать схемы существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих атрибутов. Однако в большинстве существующих систем такая возможность не допускается, и хотя упорядоченность набора атрибутов отношения явно не требуется, часто в качестве неявного порядка атрибутов используется их порядок в линейной форме определения схемы отношения.

4.2.4 Атомарность значений атрибутов

Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого типа данных, т.е. среди значений домена не могут содержаться множества значений (отношения). Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме. Потенциальным примером ненормализованного отношения является следующее:

НОМЕР_ОТДЕЛА	ОТДЕЛ		
	СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП
310	2934	Иванов	112,000
	2935	Петров	112,500
313	2937	Федоров	110,000
315	2938	Иванова	112,000

Можно сказать, что здесь мы имеем бинарное отношение, значениями атрибута ОТДЕЛЫ которого являются отношения. Представим нормализованный вариант исходного отношения:

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	НОМЕР_ОТДЕЛА
2934	Иванов	112,000	310
2935	Петров	144,000	310
2936	Сидоров	92,000	313
2937	Федоров	110,000	310
2938	Иванова	112,000	315

Нормализованные отношения составляют основу классического реляционного подхода к организации баз данных. Они обладают некоторыми ограничениями (не любую информацию удобно представлять в виде плоских таблиц), но существенно упрощают манипулирование данными. Рассмотрим, например, два идентичных оператора занесения кортежа:

- Зачислить сотрудника Кузнецова (пропуск номер 3000, зарплата 115,000) в отдел номер 320 и
- Зачислить сотрудника Кузнецова (пропуск номер 3000, зарплата 115,000) в отдел номер 310.

Если информация о сотрудниках представлена в виде нормализованного отношения, оба оператора будут выполняться одинаково (вставить кортеж в отношение). Если же работать с ненормализованным отношением, то первый оператор выразится в занесение кортежа, а второй – в добавление информации о Кузнецове в множественное значение атрибута ОТДЕЛ кортежа с первичным ключом 310.

4.3 Компоненты реляционной модели

Когда мы говорили об основных понятиях реляционных баз данных, мы не опирались на какую-либо конкретную реализацию. Эти рассуждения в равной степени относились к любой системе, при построении которой использовался реляционный подход. Другими словами, мы использовали понятия так называемой *реляционной модели данных*. Модель данных описывает некоторый набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Хотя понятие модели данных является общим, и можно говорить о иерархической, сетевой, некоторой семантической и т.д. моделях данных, нужно отметить, что это понятие было введено в обиход применительно к реляционным системам и наиболее эффективно используется именно в этом контексте.

Наиболее распространенная трактовка реляционной модели данных принадлежит Дейту, который воспроизводит ее (с различными уточнениями) практически во всех своих книгах. Согласно Дейту, реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода: *структурной части, манипуляционной части и целостной части*.

В *структурной части* модели фиксируется, что единственной структурой данных, используемой в реляционных БД, является нормализованное *n*-арное отношение. По сути дела, в предыдущих двух разделах этой лекции мы рассматривали именно понятия и свойства структурной составляющей реляционной модели.

В *манипуляционной части* модели утверждаются два фундаментальных механизма манипулирования реляционными БД – реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями), а второй – на классическом логическом аппарате исчисления предикатов первого порядка. Основной функцией манипуляционной части реляционной модели является обеспечение меры реляционности любого конкретного языка реляционных БД: язык называется реляционным, если он обладает не меньшей выразительностью и мощностью, чем реляционная алгебра или реляционное исчисление.

В *целостной части* реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД. Первое требование называется *требованием целостности сущностей*. Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно требование состоит в том, что любой кортеж любого отношения отличим от любого другого кортежа этого отношения, т.е., другими словами, любое отношение должно обладать первичным ключом. Как мы видели в предыдущем разделе, это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

Второе требование называется *требованием целостности по ссылкам* и является несколько более сложным. Очевидно, что при соблюдении нормализованности отношений сложные сущности реального мира представляются в реляционной БД в виде нескольких кортежей нескольких отношений. Например, нам требуется представить в реляционной базе данных сущность ОТДЕЛ с атрибутами ОТД_НОМЕР (номер отдела), ОТД_КОЛ (количество сотрудников) и ОТД_СОТР (набор сотрудников отдела). Для каждого сотрудника нужно хранить СОТР_НОМЕР (номер сотрудника), СОТР_ИМЯ (имя сотрудника) и СОТР_ЗАРП (заработная плата сотрудника). При правильном проектировании соответствующей БД в ней появятся два отношения ОТДЕЛЫ и СОТРУДНИКИ со схемами:

$S_{ОТДЕЛЫ}$ = (ОТД_НОМЕР, ОТД_КОЛ),
где первичный ключ – ОТД_НОМЕР и
$S_{СОТРУДНИКИ}$ = (СОТР_НОМЕР, СОТР_ИМЯ, СОТР_ЗАРП,
НОМЕР_ОТДЕЛА),
где первичный ключ – СОТР_НОМЕР.

Как видно, атрибут НОМЕР_ОТДЕЛА появляется в отношении СОТРУДНИКИ не потому, что номер отдела является собственным свойством сотрудника, а лишь для того, чтобы иметь возможность восстановить при необходимости полную сущность ОТДЕЛ. Значение атрибута НОМЕР_ОТДЕЛА в любом кортеже отношения СОТРУДНИКИ должно соответствовать значению атрибута ОТД_НОМ в некотором кортеже отношения ОТДЕЛЫ. Атрибут такого рода называется *внешним ключом*, поскольку его значения однозначно характеризуют сущности, представленные кортежами некоторого другого отношения (т.е. задают значения их первичного ключа). Говорят, что отношение, в котором определен внешний ключ, ссылается на соответствующее отношение, в котором такой же атрибут является первичным ключом.

Требование целостности по ссылкам, или требование внешнего ключа состоит в том, что для каждого значения внешнего ключа, появляющегося в ссылающемся отношении, в отношении, на которое ведет ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть неопределенным (т.е. ни на что не указывать). Для нашего примера это означает, что если для сотрудника указан номер отдела, то этот отдел должен существовать.

Ограничения целостности сущности и по ссылкам должны поддерживаться СУБД. Для соблюдения целостности сущности достаточно гарантировать отсутствие в любом отношении кортежей с одним и тем же значением первичного ключа. С целостностью по ссылкам дела обстоят несколько более сложно.

Понятно, что при обновлении ссылающегося отношения (вставке новых кортежей или модификации значения внешнего ключа в существующих кортежах) достаточно следить за тем, чтобы не появлялись некорректные значения внешнего ключа. Но как быть при удалении кортежа из отношения, на которое ведет ссылка?

Здесь используются три стратегии поддержки целостности по ссылкам, определенные для соответствующих сущностей инфологической модели БД (п. 2.5): каскадирование, ограничение, установка.

В развитых реляционных СУБД можно выбрать способ поддержания целостности по ссылкам для каждого внешнего ключа. Конечно, для принятия такого решения необходимо анализировать требования конкретной прикладной области.

4.4 Реляционная алгебра

Основная идея реляционной алгебры состоит в том, что коль скоро отношения являются множествами, то средства манипулирования отношениями могут базироваться на традиционных теоретико-множественных операциях, дополненных некоторыми специальными операциями, специфичными для баз данных.

Существует много подходов к определению реляционной алгебры, которые различаются набором операций и способами их интерпретации, но в принципе более или менее равносильны. Мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом. В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса – *теоретико-множественные операции* и *специальные реляционные операции*.

В состав теоретико-множественных операций входят операции:

1) объединения отношений;
2) пересечения отношений;
3) взятия разности отношений;
4) Декартова (прямого) произведения отношений.

Специальные реляционные операции включают:

1) ограничение (выборка, фильтрация) отношения;
2) проекцию отношения;
3) условное соединение отношений;
4) деление отношений.

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 заголовок (схему) результирующего отношения.

4.4.1 Реляционные операции

Для описания результатов операций реляционной алгебры будем использовать нотацию теории множеств. Следующая запись означает, что множество R состоит из элементов r:

$$R = \{r \mid \text{условие}\} ,$$

где «условие» определяет принадлежность элемента r к множеству R.

Мы будем рассматривать множества-отношения, состоящие из элементов-кортежей. Условие – это логическое выражение, которое может включать логические операции $\wedge$ -»И» и $\vee$-»ИЛИ».

При выполнении операции *объединения* отношений R и Q производится отношение, включающее все кортежи r, входящие хотя бы в одно из отношений-операндов:

$$R \cup Q = \{r \mid r \in R \vee r \in Q\} .$$

Операция *пересечения* отношений R и Q производит отношение, включающее все кортежи r, входящие в оба отношения-операнда:

$$R \cap Q = \{r \mid r \in R \wedge r \in Q\} .$$

Отношение, являющееся *разностью* отношений R и Q, включает все кортежи r, входящие в R, и не входит в Q:

$$R \setminus Q = \{r \mid r \in R \wedge r \notin Q\}.$$

Очевидно, что $R \setminus Q \neq Q \setminus R$. Заметим, также, что если $R \setminus Q = \varnothing$, то $R \subseteq Q$. Поэтому разность можно использовать для проверки: является ли R подмножеством Q.

Следует отметить, что операции объединения, пересечения и разности определены *только для эквивалентных отношений*. В противном случае найти пересечение либо разность отношений невозможно, а результат объединения не будет являться отношением.

При выполнении *Декартова произведения* отношений $R = \{r\}$ и $Q = \{q\}$ производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов (r,q):

$$R \oplus Q = \{(r, q) \mid r \in R \wedge q \in Q\}.$$

Результатом *ограничения (выборки или фильтрации)* отношения R по некоторому условию α(r) является отношение, включающее кортежи r, удовлетворяющие этому условию:

$$R[\alpha(r)] = \{r \mid r \in R \wedge \alpha(r) = True\}.$$

Пусть S – множество атрибутов отношения R, и множество A является подмножеством S: $A \subseteq S$. При выполнении *проекции* отношения на множество его атрибутов A производится отношение со схемой, состоящей из множества атрибутов A, кортежи которого производятся путем удаления из них значений, не принадлежащих атрибутам из множества A:

$$R[A] = \{r[A]\}.$$

При *условном соединении* отношений $R = \{r\}$ и $Q = \{q\}$ по некоторому условию α(r,q) производится отношение, кортежи которого являются конкатенацией кортежей (r,q) и удовлетворяют этому условию:

$$R[\alpha(r, q)]Q = \{(r, q) \mid r \in R \wedge q \in Q \wedge \alpha(r, q) = True\}.$$

Очевидно, что условное соединение сводится к последовательному применению Декартова произведения и выборки (фильтрации).

Пусть A_R и A_Q – множества атрибутов отношений R и Q, соответственно. Предположим, что

- $A \subseteq A_R$, $B \subseteq A_Q$;
- $A^1 \subseteq A_R$, $A^1 \cup A = A_R$, $A^1 \cap A = \varnothing$, т.е. A^1 состоит из всех атрибутов из A_R, не вошедших в A.

Множеством образов кортежа x проекции R[A] называют множество таких кортежей y проекции $R[A^1]$, для которых сцепление (x,y) является кортежем отношения R:

$$Im[A(x)] = \{y \mid y \in R[A^1] \wedge (x, y) \in R\}.$$

Например: Im[НОМЕР_ОТДЕЛА(2)] для отношения СОТРУДНИКИ – это отношение со схемой (СОТР_НОМЕР, СОТР_ИМЯ, СОТР_ЗАРП), кортежи которого содержат сведения о сотрудниках отдела № 2.

Предположим, что схемы проекций R[A] и Q[B] эквивалентны: $S_{R[A]} \sim S_{Q[B]}$. Операция *деления* R на Q по наборам атрибутов A и B производит отношение, состоящее из тех кортежей проекции $R[A^1]$, для которых проекция Q[B] является подмножеством множества образов кортежа:

$$R[A:B]Q = \{x \mid x \in R[A^1] \wedge Q[B] \subseteq Im[A^1(x)]\}.$$

Операция деления является наиболее сложной, и ее выполнение сводится к более простым операциям. Возможна следующая схема выполнения деления. Во-первых, находятся проекции $R[A^1]$ и Q[B]. Затем, для каждого картежа x из $R[A^1]$ строится множество образов $Im[A^1(x)]$ путем выборки из R по условию совпадения значений атрибутов и проекции R[A]. Далее, из $R[A^1]$ производится выборка по условию $Q[B] \subseteq Im[A^1(x)]$. Для проверки этого условия выполняется нахождение разности, т.к. условие эквивалентно следующему: $Q[B] \setminus Im[A^1(x)] = \varnothing$.

Предположим, что в базе данных сотрудников поддерживаются два отношения: СОТРУДНИКИ(ИМЯ, ОТД_НОМЕР) и ИМЕНА(ИМЯ), причем унарное отношение ИМЕНА содержит все фамилии, которыми обладают сотрудники организации. Тогда после выполнения операции реляционного деления отношения СОТРУДНИКИ на отношение ИМЕНА будет получено унарное отношение, содержащее номера отделов, сотрудники которых обладают всеми возможными в этой организации именами.

Операция *переименования* производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.

Операция *присваивания* позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

Поскольку результатом любой реляционной операции (кроме операции присваивания) является некоторое отношение, можно образовывать *реляционные выражения*, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение.

4.4.2 Замкнутость реляционной алгебры

Каждое отношение характеризуется схемой (или заголовком) и набором кортежей (или телом). Поэтому, если действительно желать иметь алгебру, операции которой замкнуты относительно понятия отношения, то каждая операция должна производить отношение в полном смысле, т.е. оно должно обладать и телом, и заголовком. Только в этом случае будет действительно возможно строить вложенные выражения.

Заголовок отношения представляет собой множество пар (имя-атрибута, имя-домена). Если посмотреть на определения реляционных операций, то видно, что домены атрибутов результирующего отношения однозначно определяются доменами отношений-операндов. Однако с именами атрибутов результата не всегда все так просто.

Например, представим себе, что у отношений-операндов операции Декартова произведения имеются одноименные атрибуты с одинаковыми доменами. Каким был бы заголовок результирующего отношения? Поскольку это множество, в нем не должны содержаться одинаковые элементы. Но и потерять атрибут в результате недопустимо. А это значит, что в этом случае вообще невозможно корректно выполнить операцию Декартова произведения.

Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для их разрешения в состав операций реляционной алгебры вводится операция переименования. Ее следует применять в любом случае, когда возникает конфликт именования атрибутов в отношениях – операндах одной реляционной операции. Тогда к одному из операндов сначала применяется операция переименования, а затем основная операция выполняется уже безо всяких проблем.

В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных случаях.

4.4.3 Особенности теоретико-множественных операций

Хотя в основе теоретико-множественной части реляционной алгебры лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями.

Начнем с операции объединения (все, что будет говориться по поводу объединения, переносится на операции пересечения и взятия разности). Смысл операции объединения в реляционной алгебре в целом остается

теоретико-множественным. Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если допустить в реляционной алгебре возможность теоретико-множественного объединения произвольных двух отношений (с разными схемами), то, конечно, результатом операции будет множество, но множество разнотипных кортежей, т.е. не отношение. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Все эти соображения приводят к появлению понятия *совместимости отношений по объединению*: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. Более точно это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене.

Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности результатом операции является отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения «почти» совместимы по объединению, т.е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа соединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является очевидно избыточным, поскольку известно, что любая из этих операций выражается через две других. Тем не менее, Кодд решил включить все три операции, исходя из интуитивных потребностей потенциального пользователя системы реляционных БД, далекого от математики.

Другие проблемы связаны с операцией взятия Декартова произведения двух отношений. В теории множеств Декартово произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Поскольку отношения являются множествами, то и для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут являться не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специальная форма операции взятия Декартова произведения – *расширенное Декартово произведение* отношений. При взятии расширенного Декартова произведения двух отношений элементом результирующего отношения является кортеж, представляющий собой *конкатенацию (или слияние)* одного кортежа первого отношения и одного кортежа второго отношения.

Но теперь возникает второй вопрос – как получить корректно сформированный заголовок отношения-результата? Очевидно, что проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Эти соображения приводят к появлению понятия *совместимости по взятию расширенного Декартова произведения*. Два отношения совместимы по взятию прямого произведения в том и только в том случае, если множества имен атрибутов этих отношений не пересекаются. Любые два отношения могут быть сделаны совместимыми по взятию прямого произведения путем применения операции переименования к одному из этих отношений.

Следует заметить, что операция взятия прямого произведения не является слишком осмысленной на практике. Во-первых, мощность ее результата очень велика даже при допустимых мощностях операндов, а во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как мы увидим немного ниже, основной смысл включения операции расширенного прямого произведения в состав реляционной алгебры состоит в том, что на ее основе определяется действительно полезная операция соединения.

По поводу теоретико-множественных операций реляционной алгебры следует еще заметить, что все четыре операции являются *ассоциативными*. То есть, если обозначить через OP любую из четырех операций, то (A OP B) OP C = A (B OP C), и следовательно, без введения двусмысленности можно писать A OP B OP C (A, B и C – отношения, обладающие свойствами, требуемыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т.е. A OP B = B OP A.

Несмотря на то, что теоретико-множественные операции довольно просты, их комбинации позволяют решать достаточно сложные задачи.

Примеры.

Пусть для учета абитуриентов во время сдачи вступительных экзаменов используется база данных, содержащая три отношения с эквивалентными схемами:

`R1(ФИО, Паспорт, Школа)` – список абитуриентов, сдававших репетиционный экзамен;

`R2(ФИО, Паспорт, Школа)` – список абитуриентов, сдававших вступительный экзамен;

`R3(ФИО, Паспорт, Школа)` – список абитуриентов, зачисленных в вуз.

Зачисление в вуз возможно как по результатам репетиционного, так и вступительного экзамена. Тогда:

R1∩R2\R3 – список абитуриентов, сдававших экзамены дважды, но не поступивших в вуз;

(R1\R2∩R3)∪(R2\R1∩R3) – список абитуриентов, поступивших с первой попытки;

R1∩R2∩R3 – список абитуриентов, поступивших со второй попытки.

4.4.4 Особенности специальных реляционных операций

Рассмотрим подробнее специальные реляционные операции реляционной алгебры: ограничение, проекция, соединение и деление.

Операция ограничения требует наличия двух операндов: ограничиваемого отношения и простого условия ограничения. Простое условие ограничения может иметь либо вид (a comp-op b), где a и b – имена атрибутов ограничиваемого отношения, для которых осмысленна операция сравнения comp-op, либо вид (a comp-op const), где a – имя атрибута ограничиваемого отношения, a const – литерально заданная константа.

В результате выполнения операции ограничения производится отношение, заголовок которого совпадает с заголовком отношения-операнда, а в тело входят те кортежи отношения-операнда, для которых значением условия ограничения является true.

Пусть UNION обозначает операцию объединения, INTERSECT – операцию пересечения, а MINUS – операцию взятия разности. Для обозначения операции ограничения будем использовать конструкцию A WHERE comp, где A – ограничиваемое отношение, а comp - простое условие сравнения. Пусть comp1 и comp2 – два простых условия ограничения. Тогда по определению:

A WHERE comp1 AND comp2 обозначает то же самое, что и (A WHERE comp1) INTERSECT (A WHERE comp2)

A WHERE comp1 OR comp2 обозначает то же самое, что и (A WHERE comp1) UNION (A WHERE comp2)

A WHERE NOT comp1 обозначает то же самое, что и A MINUS (A WHERE comp1)

С использованием этих определений можно использовать операции ограничения, в которых условием ограничения является произвольное булевское выражение, составленное из простых условий с использованием логических связок AND, OR, NOT и скобок.

На интуитивном уровне операцию ограничения лучше всего представлять как взятие некоторой «горизонтальной» вырезки из отношения-операнда.

Операция взятия проекции также требует наличия двух операндов – проецируемого отношения A и списка имен атрибутов, входящих в заголовок отношения A.

Результатом проекции отношения A по списку атрибутов a_1, a_2, ..., a_n является отношение, с заголовком, определяемым множеством атрибутов a_1, a_2, ..., a_n, и с телом, состоящим из кортежей вида $<a_1:v_1, a_2:v_2, ..., a_n:v_n>$ таких, что в отношении A имеется кортеж, атрибут a_1 которого имеет значение v_1, атрибут a_2 – значение v_2, ..., атрибут a_n – значение v_n. Тем самым, при выполнении операции проекции выделяется «вертикальная» вырезка отношения-операнда с естественным *уничтожением потенциально возникающих кортежей-дубликатов*.

Общая операция условного соединения (называемая также соединением по условию) требует наличия двух операндов – соединяемых отношений и третьего операнда – простого условия. Пусть соединяются отношения A и B. Как и в случае операции ограничения, условие соединения comp имеет вид либо (a comp-op b), либо (a comp-op const), где a и b – имена атрибутов отношений A и B, const – литерально заданная константа, а comp-op – допустимая в данном контексте операция сравнения.

Тогда по определению результатом операции условного соединения является отношение, получаемое путем выполнения операции ограничения по условию comp Декартова произведения отношений A и B.

Если внимательно осмыслить это определение, то станет ясно, что в общем случае применение условия соединения существенно уменьшит мощность результата промежуточного прямого произведения отношений-операндов только в том случае, когда условие соединения имеет вид (a comp-op b), где a и b – имена атрибутов разных отношений-операндов. Поэтому на практике обычно считают реальными операциями соединения именно те операции, которые основываются на условии соединения приведенного вида.

Хотя операция соединение в нашей интерпретации не является примитивной (поскольку она определяется с использованием прямого произведения и проекции), в силу особой практической важности она включается в базовый набор операций реляционной алгебры. Заметим также, что в практических реализациях соединение обычно не выполняется именно как ограничение прямого произведения. Имеются более эффективные алгоритмы, гарантирующие получение такого же результата.

Имеется важный частный случай соединения – эквисоединение и простое, но важное расширение операции эквисоединения – естественное соединение. Операция соединения называется *операцией эквисоединения*, если условие соединения имеет вид (a = b), где a и b – атрибуты разных операндов соединения. Этот случай важен потому, что (a) он часто встречается на практике и (b) для него существуют эффективные алгоритмы реализации.

Операция *естественного соединения* применяется к паре отношений A и B, обладающих (возможно составным) общим атрибутом *c* (т.е. атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть ab обозначает объединение заголовков отношений A и B. Тогда естественное соединение A и B – это спроектированный на ab результат эквисоединения A и B. Если вспомнить введенное нами в конце предыдущей главы определение внешнего ключа отношения, то должно стать понятно, что основной смысл операции естественного соединения – возможность *восстановления сложной сущности*, декомпозированной по причине требования первой нормальной формы. Операция естественного соединения не включается прямо в состав набора операций реляционной алгебры, но она имеет очень важное практическое значение.

5 ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БД

При проектировании базы данных решаются две основные проблемы:

- Каким образом отобразить объекты предметной области в абстрактные объекты модели данных, чтобы это отображение не противоречило семантике предметной области и было по возможности лучшим (эффективным, удобным и т.д.)? Часто эту проблему называют проблемой логического проектирования баз данных.
- Как обеспечить эффективность выполнения запросов к базе данных, т.е. каким образом, имея в виду особенности конкретной СУБД, расположить данные во внешней памяти, создание каких дополнительных структур (например, индексов) потребовать и т.д.? Эту проблему называют проблемой физического проектирования баз данных.

В случае реляционных баз данных трудно представить какие-либо общие рецепты по части физического проектирования. Здесь слишком много зависит от используемой СУБД. Более того, мы не будем касаться очень важного аспекта проектирования – определения ограничений целостности (за исключением ограничения первичного ключа). Дело в том, что при использовании СУБД с развитыми механизмами ограничений целостности (например, SQL-ориентированных систем) трудно предложить какой-либо общий подход к определению ограничений целостности. Эти ограничения могут иметь общий вид, и их формулировка пока относится скорее к области искусства, чем инженерного мастерства. Самое большее, что предлагается по этому поводу в литературе, это автоматическая проверка непротиворечивости набора ограничений целостности.

Так что будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том, из каких отношений должна состоять БД и какие атрибуты должны быть у этих отношений.

5.1 Получение реляционной схемы из ER-модели

Шаг 1. Каждая простая сущность превращается в таблицу. Простая сущность – сущность, не являющаяся подтипом и не имеющая подтипов. Имя сущности становится именем таблицы.

Шаг 2. Каждый атрибут становится возможным столбцом с тем же именем; может выбираться более точный формат. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, – не могут.

Шаг 3. Компоненты уникального идентификатора сущности превращаются в первичный ключ таблицы. Если имеется несколько возможных уникальных идентификатора, выбирается наиболее используемый. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно). Для именования этих столбцов используются имена концов связей и/или имена сущностей.

Шаг 4. Связи многие-к-одному (и один-к-одному) становятся внешними ключами. То есть делается копия уникального идентификатора с конца связи «один», и соответствующие столбцы составляют внешний ключ. Необязательные связи соответствуют столбцам, допускающим неопределенные значения; обязательные связи – столбцам, не допускающим неопределенные значения.

Шаг 5. Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.

Шаг 6. Если в концептуальной схеме присутствовали подтипы, то возможны два способа:

– все подтипы в одной таблице (а);
– для каждого подтипа – отдельная таблица (б).

Все в одной таблице	Таблица – на подтип
Преимущества	
Все хранится вместе Легкий доступ к супертипу и подтипам Требуется меньше таблиц	Более ясны правила подтипов Программы работают только с нужными таблицами
Недостатки	
Слишком общее решение Требуется дополнительная логика работы с разными наборами столбцов и разными ограничениями Потенциальное узкое место (в связи с блокировками) Столбцы подтипов должны быть необязательными В некоторых СУБД для хранения неопределенных значений требуется дополнительная память	Слишком много таблиц Смущающие столбцы в представлении UNION Потенциальная потеря производительности при работе через UNION Над супертипом невозможны модификации

При применении способа (а) таблица создается для наиболее внешнего супертипа, а для подтипов могут создаваться представления. В таблицу добавляется, по крайней мере, один столбец, содержащий код ТИПА; он становится частью первичного ключа.

При использовании метода (б) для каждого подтипа первого уровня (для более нижних – представления) супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы – столбцы супертипа).

Шаг 7. Имеется два способа работы при наличии исключающих связей:

– общий домен (а);
– явные внешние ключи (б).

Если остающиеся внешние ключи все в одном домене, т.е. имеют общий формат (способ (а)), то создаются два столбца: идентификатор связи и идентификатор сущности. Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения. Столбец идентификатора сущности используется для хранения значений уникального идентификатора сущности на дальнем конце соответствующей связи.

Если результирующие внешние ключи не относятся к одному домену, то для каждой связи, покрываемой дугой исключения, создаются явные столбцы внешних ключей; все эти столбцы могут содержать неопределенные значения.

Общий домен	Явные внешние ключи
Преимущества	
Нужно только два столбца	Условия соединения – явные
Недостатки	
Оба дополнительных атрибута должны использоваться в соединениях	Слишком много столбцов

5.2 Нормализация отношений

Сначала будет рассмотрен классический подход, при котором весь процесс проектирования производится в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений. Исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих лучшими свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами лучшими, чем предыдущая.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной фор-

ме, если удовлетворяет свойственному ей набору ограничений. Примером набора ограничений является ограничение первой нормальной формы – значения всех атрибутов отношения атомарны. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса-Кодда (BCNF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм:

- каждая следующая нормальная форма в некотором смысле лучше предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных свойств сохраняются.

В основе процесса проектирования лежит метод нормализации: *декомпозиция отношения*, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*. Для дальнейшего изложения нам потребуются несколько понятий.

Функциональная зависимость. В отношении R атрибут Y функционально зависит от атрибута X (X и Y могут быть составными) в том и только в том случае, если каждому значению X соответствует в точности одно значение Y:

$$R.X \rightarrow R.Y.$$

Полная функциональная зависимость. Функциональная зависимость называется полной, если атрибут Y не зависит функционально от любого подмножества X.

Взаимно независимые атрибуты. Два или более атрибута взаимно независимы, если ни один из этих атрибутов не является функционально зависимым от других.

Возможный ключ. Набор атрибутов X (составной атрибут) называется возможным ключом, если существует полная функциональная зависи-

мость R.X–>R.Y, где Y – набор всех остальных атрибутов отношения R, не вошедших в X.

Таким образом, значения возможного ключа однозначно определяют кортеж отношения R. Полная зависимость означает *минимальность ключа*: при удалении из ключа любого из атрибутов свойство однозначной идентификации кортежа теряется.

Неключевой атрибут. Неключевым атрибутом называется любой атрибут отношения, не входящий в состав ни одного возможного ключа.

Транзитивная функциональная зависимость. Функциональная зависимость R.X–>R.Y называется транзитивной, если существует такой атрибут Z (возможно, составной), что:

- $Z \not\subset X,\ Y \not\subset Z$;
- существуют зависимости R.X–>R.Z и R.Z–>R.Y;
- не существует зависимости R.Z–>R.X;

Из последнего требования следует отсутствие зависимости R.Y–>R.X. Это означает отсутствие транзитивной зависимости между возможными ключами.

Первичный ключ. Один из возможных ключей выбирается в качестве основного – *первичного ключа*. Он состоит из минимального количества атрибутов, как правило – из одного. Атрибуты первичного ключа не могут содержать неопределенных значений.

Рассмотрим следующий пример схемы отношения:

СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ(СОТРУДНИК, ЗАРПЛАТА, ОТДЕЛ, ПРОЕКТ, ЗАДАНИЕ)

Первичный ключ: (СОТРУДНИК, ПРОЕКТ).

Предположим, что существуют следующие функциональные зависимости:

СОТРУДНИК–>ЗАРПЛАТА
СОТРУДНИК–>ОТДЕЛ
ОТДЕЛ–>ЗАРПЛАТА
(СОТРУДНИК, ПРОЕКТ)–>ЗАДАНИЕ

Как видно, хотя первичным ключом является составной атрибут (СОТРУДНИК, ПРОЕКТ), атрибуты ЗАРПЛАТА и ОТДЕЛ функционально зависят от части первичного ключа, атрибута СОТРУДНИК. В результате мы не сможем вставить в отношение СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ кортеж, описывающий сотрудника, который еще не выполняет никакого проекта (первичный ключ не может содержать неопределенное значение). При удалении кортежа мы не только разрушаем связь данного сотрудника с данным проектом, но утрачиваем информацию о том, что он работает в некотором отделе. При переводе сотрудника в другой отдел мы

будем вынуждены модифицировать все кортежи, описывающие этого сотрудника, или получим несогласованный результат. Такие неприятные явления называются аномалиями схемы отношения. Они устраняются путем нормализации.

5.1.1 Вторая нормальная форма

Отношение R находится во *второй нормальной форме (2NF)* в том и только в том случае, когда находится в 1NF, и каждый неключевой атрибут полностью зависит от первичного ключа. В этом определении предполагается, что единственным возможным ключом отношения является первичный ключ.

Произведем следующую декомпозицию отношения СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ в два отношения СОТРУДНИКИ-ОТДЕЛЫ и СОТРУДНИКИ-ПРОЕКТЫ:

СОТРУДНИКИ-ОТДЕЛЫ(СОТРУДНИК, ЗАРПЛАТА, ОТДЕЛ)
Первичный ключ: СОТРУДНИК
Функциональные зависимости:
СОТРУДНИК–>ЗАРПЛАТА
СОТРУДНИК–>ОТДЕЛ
ОТДЕЛ–>ЗАРПЛАТА
СОТРУДНИКИ-ПРОЕКТЫ(СОТРУДНИК, ПРОЕКТ, ЗАДАНИЕ)
Первичный ключ: (СОТРУДНИК, ПРОЕКТ).
Функциональные зависимости:
(СОТРУДНИК, ПРОЕКТ)–>СОТР_ЗАДАН

Каждое из этих двух отношений находится в 2NF, и в них устранены отмеченные выше аномалии (легко проверить, что все указанные операции выполняются без проблем).

Если допустить наличие нескольких ключей, то определение 2NF примет следующий вид:

Отношение R находится во второй нормальной форме (2NF) в том и только в том случае, когда оно находится в 1NF, и каждый неключевой атрибут полностью зависит от каждого ключа R.

Здесь и далее мы не будем приводить примеры для отношений с несколькими ключами. Они слишком громоздки и относятся к ситуациям, редко встречающимся на практике.

5.1.2 Третья нормальная форма

Рассмотрим еще раз отношение СОТРУДНИКИ-ОТДЕЛЫ, находящееся в 2NF. Заметим, что функциональная зависимость СОТРУДНИК–>ЗАРПЛАТА является транзитивной; она является следствием функцио-

нальных зависимостей СОТРУДНИК–>ОТДЕЛ и ОТДЕЛ–>ЗАРПЛАТА. Другими словами, заработная плата сотрудника на самом деле является характеристикой не сотрудника, а отдела, в котором он работает. В результате мы не сможем занести в базу данных информацию, характеризующую заработную плату отдела, до тех пор, пока в этом отделе не появится хотя бы один сотрудник (первичный ключ не может содержать неопределенное значение). При удалении кортежа, описывающего последнего сотрудника данного отдела, мы лишимся информации о заработной плате отдела. Чтобы согласованным образом изменить заработную плату отдела, мы будем вынуждены предварительно найти все кортежи, описывающие сотрудников этого отдела. То есть в отношении СОТРУДИКИ-ОТДЕЛЫ по-прежнему существуют аномалии. Их можно устранить путем дальнейшей нормализации.

Отношение R находится в *третьей нормальной форме (3NF)* в том и только в том случае, если находится во 2NF и не содержит транзитивных зависимостей.

Можно произвести декомпозицию отношения СОТРУДНИКИ-ОТДЕЛЫ в два отношения СОТРУДНИКИ и ОТДЕЛЫ:

СОТРУДНИКИ (СОТРУДНИК, ОТДЕЛ)
Первичный ключ: СОТРУДНИК
Функциональные зависимости:
СОТРУДНИК–>ОТДЕЛ
ОТДЕЛЫ(ОТДЕЛ, ЗАРПЛАТА)
Первичный ключ: ОТДЕЛ
Функциональные зависимости:
ОТДЕЛ–>ЗАРПЛАТА

Каждое из этих двух отношений находится в 3NF и свободно от отмеченных аномалий.

Если отказаться от того ограничения, что отношение обладает единственным ключом, то определение 3NF примет следующую форму:

Отношение R находится в третьей нормальной форме (3NF) в том и только в том случае, если находится во 2NF, и каждый неключевой атрибут не является транзитивно зависимым от какого-либо ключа R.

На практике третья нормальная форма схем отношений достаточна в большинстве случаев, и приведением к третьей нормальной форме процесс проектирования реляционной базы данных обычно заканчивается. Однако иногда полезно продолжить процесс нормализации.

5.1.3 Нормальная форма Бойса-Кодда

Рассмотрим следующий пример схемы отношения:

СОТРУДНИКИ-ПРОЕКТЫ (СОТР_НОМЕР, СОТР_ФИО, ПРОЕКТ, ЗАДАНИЕ)

Возможные ключи: (СОТР_НОМЕР, ПРОЕКТ), (СОТР_ФИО, ПРОЕКТ)

Функциональные зависимости:
(СОТР_НОМЕР, ПРОЕКТ)–>СОТР_ЗАДАН
(СОТР_ФИО, ПРОЕКТ)–>СОТР_ЗАДАН
СОТР_НОМЕР–>ПРОЕКТ
СОТР_ФИО–>ПРОЕКТ
СОТР_НОМЕР–>СОТР_ФИО
СОТР_ФИО–>СОТР_НОМЕР

В этом примере мы предполагаем, что личность сотрудника полностью определяется как его номером, так и именем (это не вполне корректное предположение, но достаточное для примера).

Отношение СОТРУДНИКИ-ПРОЕКТЫ находится в 3NF. Две последние зависимости означают, что имеются неполные функциональные зависимости от возможного ключа атрибута, который является частью другого возможного ключа. Это также приводит к аномалиям. Например, для того чтобы изменить имя сотрудника с данным номером согласованным образом, нам потребуется модифицировать все кортежи, включающие его номер.

Детерминант. Детерминантом называется любой атрибут, от которого полностью функционально зависит некоторый другой атрибут.

Отношение R находится в *нормальной форме Бойса-Кодда (BCNF)* в том и только в том случае, если оно находится в 3NF и каждый детерминант является возможным ключом.

Очевидно, что это требование не выполнено для отношения СОТРУДНИКИ-ПРОЕКТЫ. Можно произвести его декомпозицию к отношениям СОТРУДНИКИ и СОТРУДНИКИ-ПРОЕКТЫ:

СОТРУДНИКИ(СОТР_НОМЕР, СОТР_ФИО)
Возможные ключи: СОТР_НОМЕР и СОТР_ФИО.
Функциональные зависимости:
СОТР_НОМЕР–>СОТР_ФИО
СОТР_ФИО–>СОТР_НОМЕР
СОТРУДНИКИ-ПРОЕКТЫ(СОТР_НОМЕР, ПРОЕКТ, ЗАДАНИЕ)
Возможный ключ: (СОТР_НОМЕР, ПРОЕКТ)
Функциональные зависимости:
(СОТР_НОМЕР, ПРОЕКТ)–>ЗАДАНИЕ

Возможна альтернативная декомпозиция, если выбрать за основу СОТР_ФИО. В обоих случаях получаемые отношения СОТРУДНИКИ и СОТРУДНИКИ-ПРОЕКТЫ находятся в BCNF, и им не свойственны отмеченные аномалии. Однако первый вариант предпочтительней, т.к. при изменении фамилии сотрудника модифицируется только один кортеж отношения СОТРУДНИКИ.

5.3 Поддержка целостности БД

Целостность (от англ. integrity – нетронутость, неприкосновенность, сохранность, целостность) – понимается как правильность данных в любой момент времени. Но эта цель может быть достигнута лишь в определенных пределах: СУБД не может контролировать правильность каждого отдельного значения, вводимого в базу данных (хотя каждое значение можно проверить на правдоподобность). Например, нельзя обнаружить, что вводимое значение 5 (представляющее номер дня недели) в действительности должно быть равно 3. С другой стороны, значение 9 явно будет ошибочным и СУБД должна его отвергнуть. Однако для этого ей следует сообщить, что номера дней недели должны принадлежать набору (1, 2, 3, 4, 5, 6, 7).

Поддержание целостности базы данных может рассматриваться как защита данных от неверных изменений или разрушений (не путать с незаконными изменениями и разрушениями, являющимися проблемой безопасности). Современные СУБД имеют ряд средств для обеспечения поддержания целостности (так же, как и средств обеспечения поддержания безопасности).

Выделяют три группы правил целостности:

- Целостность по сущностям.
- Целостность по ссылкам.
- Целостность, определяемая пользователем.

В предыдущем подразделе была рассмотрена мотивировка двух правил целостности для обозначений и характеристик, общих для любых реляционных баз данных.

Не допускается, чтобы какой-либо атрибут, участвующий в первичном ключе, принимал неопределенное значение.

Значение внешнего ключа должно либо:

- быть равным значению первичного ключа цели;
- быть полностью неопределенным, т.е. каждое значение атрибута, участвующего во внешнем ключе должно быть неопределенным.

Для любой конкретной базы данных существует ряд дополнительных специфических правил, которые относятся к ней одной и определяются разработчиком. Чаще всего контролируется:

- уникальность тех или иных атрибутов,
- диапазон значений (экзаменационная оценка от 2 до 5),
- принадлежность набору значений (пол «М» или «Ж»).

6 ФИЗИЧЕСКИЕ МОДЕЛИ ДАННЫХ

Существуют два принципиальных подхода к физическому хранению отношений. Наиболее распространенным является покортежное хранение отношений. Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одного отношения и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа. Физической единицей хранения данных является *блок (страница)*. Размер страницы определяется дисковым контроллером и операционной системой. Одна страница может содержать несколько кортежей.

Альтернативным (менее распространенным) подходом является хранение отношения по столбцам, т.е. единицей хранения является столбец отношения с исключенными дубликатами. Естественно, что при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся; за один обмен с внешней памятью в общем случае считывается больше полезной информации. Дополнительным преимуществом является возможность использования значений столбца отношения для оптимизации выполнения операций соединения. Но при этом требуются существенные дополнительные действия для сборки целого кортежа (или его части).

6.1 Индексы

Индексом называют служебную запись, предназначенную для организации доступа к основным записям файла БД. Основное назначение индекса состоит в обеспечении эффективного *прямого доступа* к кортежу отношения по ключу. Индексная запись имеет вид:

Значение ключа	Номер записи

Ключом индекса является значение атрибута (*простой ключ*) или совокупность значений нескольких атрибутов (*составной ключ*) определенного кортежа. Номер записи указывает на физическую запись, содержащую данный кортеж.

Логически файл БД можно представить состоящим из двух частей:

- индексной области, состоящей из индексных записей;
- основной области, содержащей записи БД.

Индексные записи упорядочены по значению ключа. Взаимодействие этих двух областей составляет сущность механизма индексации для ускорения доступа к данным. Физическое совмещение двух частей в одном файле не обязательно, чаще всего индексная область хранится в отдельном файле.

Если ключом индекса является возможный ключ отношения, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа отношения автоматически заводится уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа так или иначе требуется индексная поддержка.

Поскольку при выполнении многих операций языкового уровня требуется сортировка отношений в соответствии со значениями некоторых атрибутов, полезным свойством индекса является обеспечение последовательного просмотра кортежей отношения в диапазоне значений ключа в порядке возрастания или убывания значений ключа.

Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу и последовательный просмотр в порядке возрастания или убывания значений ключа, является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Одна организация индекса отличается от другой главным образом в способе поиска ключа с заданным значением.

Время доступа к записи оценивается в количествах обращений к устройству внешней памяти (диску), т.к. работа с диском является наиболее длительной операцией по сравнению с манипуляциями в оперативной памяти. При таком подходе время доступа определяется целиком только способом доступа и не зависит от быстродействия конкретного накопителя.

6.1.1 Плотные индексы

В файлах с *плотным индексом* (*индексно-прямые файлы*) основная область содержит последовательность записей одинаковой длины, расположенных в произвольном порядке. Значением ключа индексной записи является значение первичного ключа отношения, поэтому ключ однозначно определяет основную запись. Таким образом, каждой записи в основной области соответствует одна запись из индексной области. Все записи индексной области упорядочены по значению ключа, что позволяет применять эффективные способы поиска записи.

Наиболее эффективным алгоритмом поиска на упорядоченном массиве является *бинарный поиск*, основанный на методе деления пополам:

1) все пространство поиска делится пополам;
2) выбирается элемент, расположенный посредине;
3) если данный элемент является искомым, поиск завершается;
4) если нет, определяется, в какой из половин он должен находится;
5) новым пространством поиска выбирается соответствующая половина и переходят к шагу 1.

Если пространство поиска состоит из N элементов, то максимальное количество шагов поиска будет равно:

$$T_{\max} = \log_2 N .$$

Оценим максимальное количество обращений к дисковой памяти при поиске записи. На диске записи хранятся в физических блоках, размер которых определяется контроллером диска. Один блок может содержать несколько индексных записей. За одно обращение к диску считывается один блок. Для поиска записи необходимо найти требуемый индексный блок, прочесть требуемую индексную запись, прочесть основной блок с требуемой записью. Если файл содержит N_i индексных блоков, то *максимальное время доступа* составит:

$$T_{\max} = \log_2 N_i + 1.$$

При последовательном доступе (без использования индекса) максимальное время доступа $T_{\max}=N_i$. Например, если $N_i=1024$, то при последовательном доступе нужно 1024 обращений к диску, а с использованием индекса – 10+1=11!

При добавлении новой записи она присоединяется в конец основной области. Соответствующая ей индексная запись должна быть помещена в строго определенной место, т.к. индексные записи упорядочены. Для этого индексные блоки содержат свободные области, первоначальный размер которых (*процент расширения*) определяется СУБД. Для добавления индексной записи необходимо:

- найти соответствующий индексный блок;
- прочесть его в оперативную память и модифицировать – вставить новую запись;
- записать блок обратно на диск.

Таким образом, время добавления новой записи составит:

$$T_{\max} = \log_2 N_i + 1 + 1 + 1.$$

При добавлении записей процент расширения индексных блоков уменьшается. При отсутствии в блоке свободной области (*переполнение*) возможны два решения:

1) перестройка индексной области;
2) организация дополнительной области переполнения для не помещающихся индексных записей.

Первый способ более трудоемкий, второй приводит к увеличению времени доступа к записям.

При удалении записи необходимо:

- найти соответствующий индексный блок;

– прочесть его в оперативную память и модифицировать – удалить запись с переписыванием всех последующих на ее место;
– записать блок обратно на диск;
– пометить соответствующую основную запись как удаленную.

Процент расширения индексного блока при удалении записи увеличивается, для удаления требуется такое же количество обращений к диску, как и при записи.

6.1.2 Неплотные индексы

В файлах с *неплотным индексом* (*индексно-последовательные файлы*) основная область содержит упорядоченную последовательность записей одинаковой длины. Значением ключа индексной записи является значение первичного ключа первой записи в основном блоке:

Ключ первой записи блока	**Номер блока**

Структура индексно-последовательного файла представлена на рисунке 6.1.

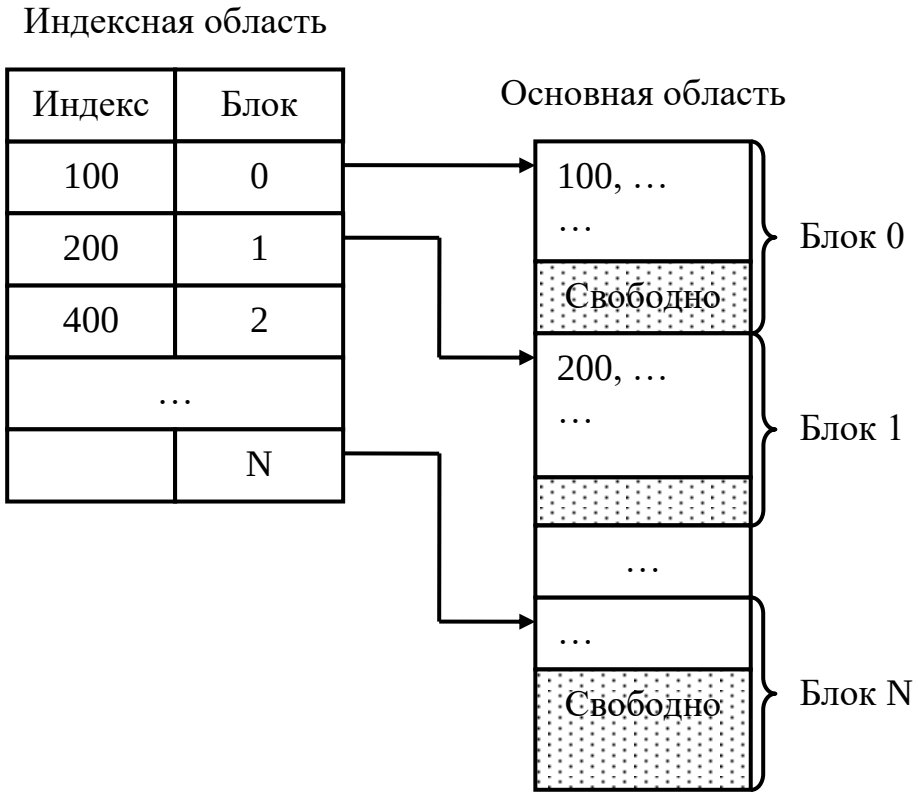


Рис. 6.1. Структура индексно-последовательного файла

Все записи индексной области по-прежнему упорядочены по значению ключа. Каждая запись из индексной области соответствует блоку в основной области. Таким образом, упорядоченное хранение основных записей позволяет уменьшить количество хранимых индексных записей. Это приводит к уменьшению количества индексных блоков N_i и, соответственно, времени доступа по сравнению с плотным индексом.

Новая запись должна заноситься в строго определенный основной блок на соответствующее место. Поэтому алгоритм добавления записи в индексно-последовательный файл состоит из следующих шагов:

- найти соответствующий основной блок;
- прочесть его в оперативную память и модифицировать – вставить новую запись на ее место;
- записать блок обратно на диск.

Индексная область при добавлении новой записи не корректируется. Процент расширения для блока основной области (Full-factor) определяется СУБД, при переполнении блока происходит перестройка всего индексно-последовательного файла.

При удалении записи она физически уничтожается в основной области, при этом индексная область обычно не корректируется, даже если удаляется первая запись блока. Поэтому количество обращений к диску такое же, как при добавлении новой записи.

Пример. Предположим, что БД содержит 2^{18}=262144 записей длиной 2^7=128 байт; размер блока – 2^{10}=1024 байт (8 записей в блоке); размер ключа – 5 байт. Тогда для хранения записей БД потребуется 262144/8 = 32768 основных блоков.

При использовании плотного индекса для ссылки на 262144 записей необходимо поле номера записи длиной 3 байта. Тогда длина индексной записи равна 5+3 = 8 байт и индексный блок будет содержать 1024/8 = 128 записей. Всего потребуется 262144/128 = 2048 индексных блоков. Максимальное время доступа составит:

$$\log_2 2048 + 1 = 11 + 1 = 12.$$

При неплотном индексе для ссылки на 32768=2^{15} основных блока необходимо поле номера блока длиной 2 байта (2^{16}). Тогда длина индексной записи составит 5+2 = 7 байт и индексный блок будет содержать 1024/7 = 147 записей. Всего потребуется 32768/147 = 223 индексных блока. Максимальное время доступа:

$$\log_2 223 + 1 = 8 + 1 = 9.$$

Таким образом, использование неплотного индекса сокращает время доступа на 25 %.

6.2 B-деревья

Наиболее популярным подходом к организации индексов в базах данных является использование техники B-деревьев. Она основана на мультииндексировании – применении индексирования для ускорения доступа к индексным блокам.

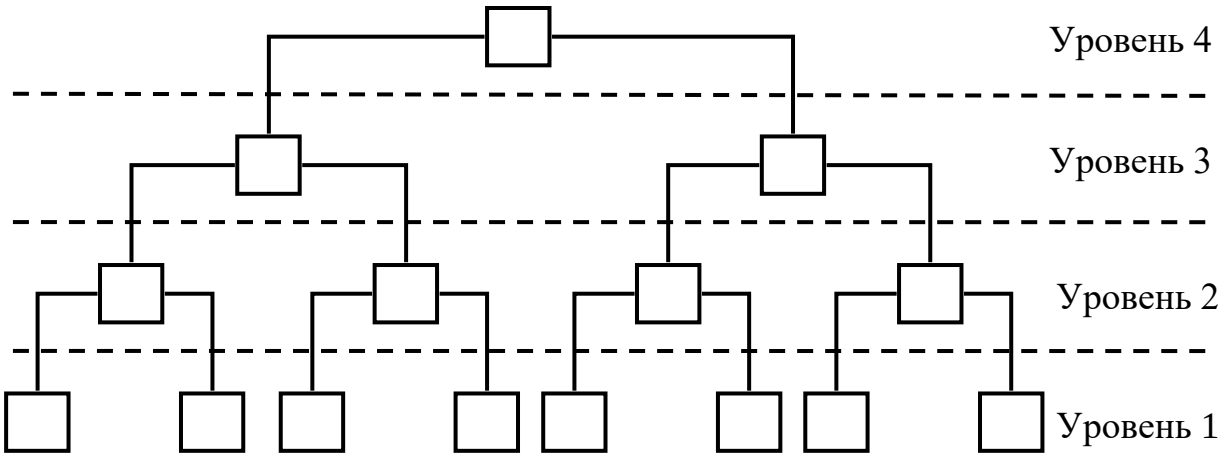


Рис. 6.2. B-дерево

Действительно, сама индексная область может рассматриваться как основной файл, для которого можно построить неплотный индекс. Над новым индексом строится следующий, и так до тех пор, пока мы не получим индексную область, состоящую из одного блока. В итоге мы будем иметь дерево, представленное на рисунке 6.2, у которого:

- внутренние узлы соответствуют блокам, содержащим индексные записи для индексов;
- листовые узлы соответствуют блокам, содержащим индексные записи для основных записей;
- узел-родитель содержит индексы для узлов-потомков.

Поиск в B-дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. При этом *длина пути* (количество проходимых узлов) от корня дерева к любому его листу одна и та же. Такое дерево относится к категории *сбалансированных (balanced) деревьев* или *B-деревьев*.

Заметим, что поскольку деревья сильно ветвистые и сбалансированные, то для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутреннем блоке помещается n ключей, то при хранении m записей требуется дерево глубиной $\log_n(m)$, где $\log_n$ вычисляет логарифм по основанию n. Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск.

Основной «изюминкой» B-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим в качестве примера операции занесения и удаления записей.

При занесении новой записи выполняется:

– Поиск листового блока. Фактически производится обычный поиск по ключу. Если в B-дереве не содержится ключ с заданным значением, то будет получен номер блока, в которой ему надлежит содержаться, и соответствующие координаты внутри блока.
– Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовой блок, в который требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер блока внешней памяти.

Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера блока, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память, или временно сохранен в оперативной памяти в зависимости от политики управления буферами.

Если же возникло переполнение блока (т.е. размер используемой части буфера превосходит размер блока), то выполняется расщепление блока. Для этого запрашивается новый блок внешней памяти, используемая часть буфера разбивается грубо говоря пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенный блок, а в старом блоке модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых блоков.

Чтобы обеспечить доступ от корня дерева к заново заведенному блоку, необходимо соответствующим образом модифицировать внутренний блок, являющийся предком ранее существовавшего листового блока, т.е. вставить в него соответствующее значение ключа и ссылку на новый блок. При выполнении этого действия может снова произойти переполнение теперь уже внутреннего блока, и он будет расщеплен на два. В результате потребуется вставить значение ключа и ссылку на новый блок во внутренний блок-предка выше по иерархии и т.д.

Предельным случаем является переполнение корневого блока B-дерева. В этом случае он тоже расщепляется на два, и заводится новый корневой блок дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

– Поиск записи по ключу. Если запись не найдена, то значит удалять ничего не нужно.
– Реальное удаление записи в буфере, в который прочитан соответствующий листовой блок.

Если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых блоках, являющихся левым или правым братом данного блока, больше, чем размер блока, операция завершается.

Иначе производится слияние с правым или левым братом, т.е. в буфере производится новый образ блока, содержащего общую информацию из данного блока и его левого или правого брата. Ставший ненужным листовой блок заносится в список свободных блоков. Соответствующим образом корректируется список листовых блоков.

Чтобы устранить возможность доступа от корня к освобожденному блоку, нужно удалить соответствующее значение ключа и ссылку на освобожденный блок из внутреннего блока – его предка. При этом может возникнуть потребность в слиянии этого блока с его левым или правыми братьями и т.д.

Предельным случаем является полное опустошение корневого блока дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневой блок освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности B-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

- упреждающие расщепления, т.е. расщепления блока не при его переполнении, а несколько раньше, когда степень заполненности блока достигает некоторого уровня;
- переливания, т.е. поддержание равновесного заполнения соседних блоков;
- слияния 3-в-2, т.е. порождение двух листовых блоков на основе содержимого трех соседних.

Пример. Предположим, что БД содержит 2^{18}=262144 основных блока; индексный блок содержит 2^8=256 записей, индексная область – 262144/256 = 1024 блока.

При использовании неплотного индекса максимальное время доступа составит:

$$\log_2 1024 + 1 = 10 + 1 = 11.$$

При использовании B-дерева количество блоков на уровне 1 будет равно 262144/256 = 1024; на уровне 2 будет 1024/256 = 4 блока; на уровне 3 – 4/256 = 1 блок, содержащий 4 записи – корневой блок. Таким образом,

мы будем иметь B-дерево глубиной $\log_{256}262144 = 3$ (округление в большую сторону). Тогда время доступа всегда будет равно 3 + 1 = 4.

6.3 Хэширование

Альтернативным и все более популярным подходом к организации индексов является использование техники хэширования. Общей идеей методов хэширования является применение к значению ключа некоторой функции свертки (хэш-функции), вырабатывающей значение меньшего размера. Свертка значения ключа затем используется для доступа к записи.

В самом простом, классическом случае свертка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значения свертки. При возникновении коллизий (одна и та же свертка для нескольких значений ключа) образуются цепочки переполнения. Главным ограничением этого метода является фиксированный размер таблицы. Если таблица заполнена слишком сильно или переполнена, т.е. возникнет слишком много цепочек переполнения, то главное преимущество хэширования – доступ к записи почти всегда за одно обращение к таблице – будет утрачено. Расширение таблицы требует ее полной переделки на основе новой хэш-функции (со значением свертки большего размера).

В случае баз данных такие действия являются абсолютно неприемлемыми. Поэтому обычно вводят промежуточные таблицы-справочники, содержащие значения ключей и адреса записей, а сами записи хранятся отдельно. Тогда при переполнении справочника требуется только его переделка, что вызывает меньше накладных расходов.

Чтобы избежать потребности в полной переделке справочников, при их организации часто используют технику B-деревьев с расщеплениями и слияниями. Хэш-функция при этом меняется динамически, в зависимости от глубины B-дерева. Путем дополнительных технических ухищрений удается добиться сохранения порядка записей в соответствии со значениями ключа. В целом методы B-деревьев и хэширования все более сближаются.

7 ОРГАНИЗАЦИЯ И ФУНКЦИИ РЕЛЯЦИОННЫХ СУБД

7.1 Основные функции СУБД

К числу функций СУБД принято относить следующие.

7.1.1 Непосредственное управление данными во внешней памяти

Эта функция включает обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения 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 вплоть до уровня устройств внешней памяти. 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, СУБД поддерживает собственную систему именования объектов БД.

7.1.2 Управление буферами оперативной памяти

СУБД обычно работают с БД значительного размера; по крайней мере, этот размер обычно больше доступного объема 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. При этом даже если операционная система производит общесистемную буферизацию (как 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 собственной дисциплиной замены буферов.

7.1.3 Управление транзакциями

Транзакция – это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует (COMMIT) 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 СОТРУДНИКИ и ОТДЕЛЫ, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотруд-

ника является объединение элементарных операций над файлами СОТРУДНИКИ и ОТДЕЛЫ в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД (если, конечно, такая система заслуживает названия 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 параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей 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).

7.1.4 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, поддержание надежности хранения данных в БД требует избыточности хранения данных, причем та часть данных, которая используется для восстановления, 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 (иногда поддерживаются 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.

7.1.5 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 схемы* БД *(DDL – Data Definition Language)* и *язык манипулирования данными (DML – Data Manipulation Language).* DDL служил главным образом для определения логической структуры БД, т.е. той структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными, т.е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Мы рассмотрим более подробно языки ранних СУБД в следующей лекции.

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 (Structured Query Language). Далее будут перечислены основные функции реляционной СУБД, поддерживаемые на «языковом» уровне (т.е. функции, поддерживаемые при реализации интерфейса SQL).

7.2 Типовая структура современной СУБД

Естественно, организация типичной СУБД и состав ее компонентов соответствует рассмотренному нами набору функций.

Логически в современной реляционной СУБД можно выделить:

- внутреннюю часть – ядро СУБД (часто его называют Data Base Engine);
- компилятор языка БД (обычно 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 было понять из первой части этой лекции, 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, не доступным 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 основной составляющей серверной части системы.

Основной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Основной 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. 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 выполнение 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.

7.2.1 Служебные данные

Для корректной работы подсистемы управления данными во внешней памяти необходимы данные, которые используются только этой подсистемой и не видны подсистеме языкового уровня. Набор структур служебной информации зависит от общей организации системы, но обычно требуется поддержание следующих служебных данных.

Внутренние каталоги, описывающие физические свойства объектов базы данных, например, число атрибутов отношения, их размер и, возможно, типы данных; описание индексов, определенных для данного отношения и т.д.

Описатели свободной и занятой памяти в страницах отношения. Такая информация требуется для нахождения свободного места при занесении кортежа. Отдельно приходится решать задачу поиска свободного места в случаях некластеризованных и кластеризованных отношений (в последнем случае приходится дополнительно использовать кластеризованный индекс). Как мы уже отмечали, нетривиальной является проблема освобождения страницы в условиях мультидоступа.

Связывание страниц одного отношения. Если в одном файле внешней памяти могут располагаться страницы нескольких отношений (обычно к этому стремятся), то нужно каким-то образом связать страницы одного отношения. Тривиальный способ использования прямых ссылок между страницами часто приводит к затруднениями при синхронизации транзакций (например, особенно трудно освобождать и заводить новые страницы отношения). Поэтому стараются использовать косвенное связывание страниц с использованием служебных индексов. В частности, известен общий механизм для описания свободной памяти и связывания страниц на основе B-деревьев.

7.3 Управление транзакциями

Транзакция – это неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в БД, либо воздействие всех этих операторов полностью отсутствует.

Лозунг транзакции – «Все или ничего». При завершении транзакции оператором COMMIT результаты гарантированно фиксируются во внешней памяти (смысл слова commit – «зафиксировать» результаты транзакции). При завершении транзакции оператором ROLLBACK результаты гарантированно отсутствуют во внешней памяти (смысл слова rollback – ликвидировать результаты транзакции).

Поддержание механизма транзакций – показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей во многопользовательских системах.

7.3.1 Поддержка целостности баз данных

Понятие транзакции имеет непосредственную связь с понятием целостности БД. Очень часто БД может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения БД. Например, в базе данных СОТРУДНИКИ-ОТДЕЛЫ естественным ограничением целостности является совпадения

значения атрибута ОТД_РАЗМЕР в кортеже отношения ОТДЕЛЫ, описывающем данный отдел (например, отдел 320), с числом кортежей отношения СОТРУДНИКИ, таких, что значение атрибута СОТР_ОТДЕЛ равно 320. Как в этом случае принять на работу в отдел 320 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в отношение СОТРУДНИКИ или модификация существующего кортежа в отношении ОТДЕЛЫ, после выполнения операции база данных окажется в нецелостном состоянии.

Поэтому для поддержания подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии БД и должна оставить это состояние целостными после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат (т.е. вместо оператора COMMIT выполняется оператор ROLLBACK), и БД остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Если быть более точным, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более, чем на 100,000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности – это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор ROLLBACK или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

И еще одно замечание. С точки зрения внешнего представления в момент завершения транзакции проверяются все откладываемые ограни-

чения целостности, определенные в этой базе данных. Однако при реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных СОТРУДНИКИ-ОТДЕЛЫ в ней не выполнялись операторы вставки или удаления кортежей из отношения СОТРУДНИКИ, то проверять упоминавшееся выше ограничение целостности не требуется (а проверка подобных ограничений вызывает достаточно большую работу).

7.3.2 Поддержка изолированности пользователей

Во многопользовательских системах с одной базой данных одновременно могут работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с БД в одиночку.

В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей. Действительно, если с каждым сеансом работы с базой данных ассоциируется транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.

При соблюдении обязательного требования поддержания целостности базы данных возможны следующие уровни изолированности транзакций.

Первый уровень – отсутствие потерянных изменений. Рассмотрим следующий сценарий совместного выполнения двух транзакций. Транзакция 1 изменяет объект базы данных A. До завершения транзакции 1 транзакция 2 также изменяет объект A. Транзакция 2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности). Тогда при повторном чтении объекта A транзакция 1 не видит изменений этого объекта, произведенных ранее. Такая ситуация называется ситуацией потерянных изменений. Естественно, она противоречит требованию изолированности пользователей. Во избежание такой ситуации в транзакции 1 требуется, чтобы до завершения транзакции 1 никакая другая транзакция не могла изменять объект A. Отсутствие потерянных изменений является минимальным требованием к СУБД по части синхронизации параллельно выполняемых транзакций.

Второй уровень – отсутствие чтения «грязных данных». Рассмотрим следующий сценарий совместного выполнения транзакций 1 и 2. Транзакция 1 изменяет объект базы данных A. Параллельно с этим транзакция 2 читает объект A. Поскольку операция изменения еще не завершена, транзакция 2 видит несогласованные «грязные» данные (в частности, операция транзакции 1 может быть отвернута при проверке немедленно проверяемо-

го ограничения целостности). Это тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и вправе ожидать видеть согласованные данные). Чтобы избежать ситуации чтения «грязных» данных, до завершения транзакции 1, изменившей объект A, никакая другая транзакция не должна читать объект A (минимальным требованием является блокировка чтения объекта A до завершения операции его изменения в транзакции 1).

Третий уровень – отсутствие неповторяющихся чтений. Рассмотрим следующий сценарий. Транзакция 1 читает объект базы данных A. До завершения транзакции 1 транзакция 2 изменяет объект A и успешно завершается оператором COMMIT. Транзакция 1 повторно читает объект A и видит его измененное состояние. Чтобы избежать неповторяющихся чтений, до завершения транзакции 1 никакая другая транзакция не должна изменять объект A. В большинстве систем это является максимальным требованием к синхронизации транзакций, хотя, как мы увидим немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (в частности, соответствующие операторы предусмотрены в стандарте SQL 2). Как мы уже отмечали, для поддержания целостности достаточен первый уровень. Существует ряд приложений, для которых первого уровня достаточно (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

К более тонким проблемам изолированности транзакций относится так называемая проблема кортежей-«фантомов», вызывающая ситуации, которые также противоречат изолированности пользователей. Рассмотрим следующий сценарий. Транзакция 1 выполняет оператор A выборки кортежей отношения R с условием выборки S (т.е. выбирается часть кортежей отношения R, удовлетворяющих условию S). До завершения транзакции 1 транзакция 2 вставляет в отношение R новый кортеж r, удовлетворяющий условию S, и успешно завершается. Транзакция 1 повторно выполняет оператор A, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора. Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-«фантомов», требуется более высокий «логический» уровень синхронизации транзакций. Идеи такой синхронизации (предикатные синхронизационные захваты) известны давно, но в большинстве систем не реализованы.

7.3.3 Сериализация транзакций

Понятно, что для того, чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.

План (способ) выполнения набора транзакций называется *сериальным*, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.

Сериализация транзакций – это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций, обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы их параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями могут существовать следующие *виды конфликтов*:

- W-W – транзакция 2 пытается изменять объект, измененный не закончившейся транзакцией 1;
- R-W – транзакция 2 пытается изменять объект, прочитанный не закончившейся транзакцией 1;
- W-R – транзакция 2 пытается читать объект, измененный не закончившейся транзакцией 1.

Практические методы сериализации транзакций основываются на учете этих конфликтов.

Существуют два базовых подхода к сериализации транзакций – основанный на синхронизационных захватах объектов базы данных и на использовании временных меток. Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении. Ниже мы рассмотрим эти подходы сравнительно подробно.

Предварительно заметим, что для каждого из подходов имеются две разновидности – *пессимистическая и оптимистическая*. При применении пессимистических методов, ориентированных на ситуации, когда конфликты возникают часто, конфликты распознаются и разрешаются немедленно при их возникновении. Оптимистические методы основываются на том, что результаты всех операций модификации базы данных сохраняются в рабочей памяти транзакций. Реальная модификация базы данных про-

изводится только на стадии фиксации транзакции. Тогда же проверяется, не возникают ли конфликты с другими транзакциями.

Далее мы ограничимся рассмотрением более распространенных пессимистических разновидностей методов сериализации транзакций. Пессимистические методы сравнительно просто трансформируются в свои оптимистические варианты.

7.3.4 Синхронизационные захваты

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре «клиент-сервер») является подход, основанный на соблюдении двухфазного протокола синхронизационных захватов объектов БД. В общих чертах протокол состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных *r* от имени транзакции T запрашивается синхронизационный захват объекта *r* в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных захватов являются:

- совместный режим – *S* (Shared), означающий разделяемый захват объекта и требуемый для выполнения операции чтения объекта;
- монопольный режим – *X* (eXclusive), означающий монопольный захват объекта и требуемый для выполнения операций занесения, удаления и модификации.

Захваты объектов несколькими транзакциями по чтению совместимы, т.е. нескольким транзакциям допускается читать один и тот же объект, захват объекта одной транзакцией по чтению не совместим с захватом другой транзакцией того же объекта по записи, и захваты одного объекта разными транзакциями по записи не совместимы. Правила совместимости захватов одного объекта разными транзакциями изображены в следующей таблице:

	X	S
–	да	да
X	нет	нет
S	нет	да

В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. При этом «–» соответствует состоянию объекта, для которого не установлен никакой захват. Транзакция, запросившая синхронизационный захват объекта БД, уже захваченный другой транзакцией в несовместимом режиме, блокируется до тех пор, пока захват с этого объекта не будет снят.

Заметим, что слово «нет» в нашей таблице соответствует описанным ранее возможным случаям конфликтов транзакций по доступу к объектам

базы данных (WW, RW, WR). Совместимость S-захватов соответствует тому, что конфликт RR не существует.

Для обеспечения сериализации транзакций (третьего уровня изолированности) синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов – 2PL. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции – накопление захватов;
- вторая фаза (фиксация или откат) – освобождение захватов.

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных захватов действительно обеспечивается сериализация транзакций на третьем уровне изолированности. Основная проблема состоит в том, что следует считать объектом для синхронизационного захвата?

В контексте реляционных баз данных возможны следующие альтернативы:

- файл – физический (с точки зрения базы данных) объект, область хранения нескольких отношений и, возможно, индексов;
- отношение – логический объект, соответствующий множеству кортежей данного отношения;
- страница данных – физический объект, хранящий кортежи одного или нескольких отношений, индексную или служебную информацию;
- кортеж – элементарный физический объект базы данных.

На самом деле, когда мы говорим про операции над объектами базы данных, то любая операция над кортежем фактически является и операцией над страницей, в которой этот кортеж хранится, и над соответствующим отношением, и над файлом, содержащем отношение. Поэтому действительно имеется выбор уровня объекта захвата.

Понятно, что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект – логический или физический), тем меньше синхронизационных захватов будет поддерживаться в системе, и на это, соответственно, будут тратиться меньшие накладные расходы. Более того, если выбрать в качестве уровня объектов для захватов файл или отношение, то будет решена даже проблема фантомов.

Но вся беда в том, что при использовании для захватов крупных объектов возрастает вероятность конфликтов транзакций и тем самым уменьшается допускаемая степень их параллельного выполнения. Фактически при укрупнении объекта синхронизационного захвата мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, когда на самом деле конфликтов нет.

Разработчики многих систем начинали с использования страничных захватов, полагая это некоторым компромиссом между стремлениями со-

кратить накладные расходы и сохранить достаточно высокий уровень параллельности транзакций. Но это не очень хороший выбор. Мы не будем останавливаться на деталях, но заметим, что использование страничных захватов в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организацию СУБД. В большинстве современных систем используются покортежные синхронизационные захваты.

Но при этом возникает очередной вопрос. Если единицей захвата является кортеж, то какие синхронизационные захваты потребуются при выполнении таких операций, как уничтожение отношения? Было бы довольно нелепо перед выполнением такой операции потребовать захвата всех существующих кортежей отношения. Кроме того, это не предотвратило бы возможности параллельной вставки в другой транзакции нового кортежа в уничтожаемое отношение.

Гранулированные синхронизационные захваты

Подобные рассуждения привели к разработке аппарата гранулированных синхронизационных захватов. При применении этого подхода синхронизационные захваты могут запрашиваться по отношению к объектам разного уровня: файлам, отношениям и кортежам. Требуемый уровень объекта определяется тем, какая операция выполняется (например, для выполнения операции уничтожения отношения объектом синхронизационного захвата должно быть все отношение, а для выполнения операции удаления кортежа – этот кортеж). Объект любого уровня может быть захвачен в режиме S или X.

Теперь наиболее важное отличие, на котором, собственно, держится соответствие захватов разного уровня. Вводятся специальные протоколы гранулированных захватов и новые типы захватов: перед захватом объекта в режиме S или X соответствующий объект более верхнего уровня должен быть захвачен в режиме IS, IX или SIX. Что же из себя представляют эти режимы захватов?

IS (Intented for Shared lock) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в совместном режиме. Например, при намерении читать кортежи из отношения R это отношение должно быть захвачено в режиме IS (а до этого в таком же режиме должен быть захвачен файл).

IX (Intented for eXclusive lock) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в монопольном режиме. Например, при намерении удалять кортежи из отношения R это отношение должно быть захвачено в режиме IX (а до этого в таком же режиме должен быть захвачен файл).

SIX (Shared, Intented for eXclusive lock) по отношению к некоторому составному объекту O означает совместный захват всего этого объекта с намерением впоследствии захватывать какие-либо входящие в него объек-

ты в монопольном режиме. Например, если выполняется длинная операция просмотра отношения с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего захватить это отношение в режиме SIX (а до этого захватить файл в режиме IS).

Довольно трудно описать словами все возможные ситуации. Мы ограничимся приведением полной таблицы совместимости захватов, при анализе которой можно выявить все случаи:

	X	S	IX	IS	SIX
–	да	да	да	да	да
X	нет	нет	нет	нет	нет
S	нет	да	нет	да	нет
IX	нет	нет	да	да	нет
IS	нет	да	да	да	да
SIX	нет	нет	нет	да	нет

Предикатные синхронизационные захваты

Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием захватов отношений в режимах S и X). Давно известно, что для решения этой проблемы необходимо перейти от захватов индивидуальных объектов базы данных, к захвату условий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для синхронизации уровня отношений именно потому, что отношение как логический объект представляет собой неявное условие для входящих в него кортежей. Захват отношения – это простой и частный случай предикатного захвата.

Поскольку любая операция над реляционной базой данных задается некоторым условием (т.е. в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационный захват в режиме S или X именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных захватов. Ясно, что без этого использовать предикатные захваты для синхронизации транзакций невозможно, а в общей форме проблема неразрешима.

К счастью, эта проблема сравнительно легко решается для случая простых условий. Будем называть простым условием конъюнкцию простых предикатов, имеющих вид

имя-атрибута { = > < } значение

В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистем управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию исходного оператора со сложным условием в последовательность обращений к ядру СУБД, в каждом из которых содержатся только простые условия. Следовательно, в случае типовой организации реляционной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации (рис. 7.1). Пусть R отношение с атрибутами a_1, a_2, ..., a_n, а m_1, m_2, ..., m_n – множества допустимых значений a_1, a_2, ..., a_n соответственно (все эти множества – конечные). Тогда можно сопоставить R конечное n-мерное пространство возможных значений кортежей R. Любое простое условие «вырезает» m-мерный прямоугольник в этом пространстве ($m <= n$).

Тогда *S-X*, *X-S*, *X-X* предикатные захваты от разных транзакций совместимы, если соответствующие прямоугольники не пересекаются.

Пример. Покажем, что в каких бы режимах не требовала транзакция 1 захвата условия (1<=a<=4) & (b=5), а транзакция 2 – условия (1<=a<=5) & (1<=b<=3), эти захваты всегда совместимы (n = 2).

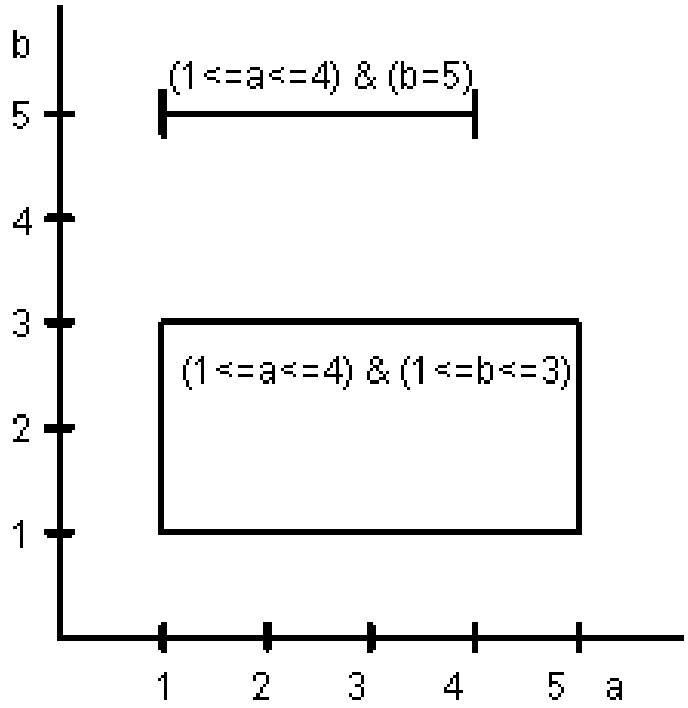


Рис. 7.1. Геометрическая интерпретация предикатных захватов

Заметим, что предикатные захваты простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов.

Тупики, распознавание и разрушение

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных захватов является возможность возникновения тупиков (deadlocks) между транзакциями. Тупики возможны при применении любого из рассмотренных нами вариантов.

Вот простой пример возникновения тупика между транзакциями T1 и T2:

- транзакции T1 и T2 установили монопольные захваты объектов r1 и r2 соответственно;
- после этого T1 требуется совместный захват r2, а T2 – совместный захват r1;
- ни одна из транзакций не может продолжаться, следовательно, монопольные захваты не будут сняты, а совместные – не будут удовлетворены.

Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) *графа ожидания транзакций*. Граф ожидания транзакций – это ориентированный двудольный граф, в котором суще-

ствует два типа вершин – вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. В этом графе существует дуга, ведущая из вершины-транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный захват объекта. В графе существует дуга из вершины-объекта к вершине-транзакции, если транзакция ожидает удовлетворения захвата объекта.

Легко показать, что в системе существует ситуация тупика, если в графе ожидания транзакций имеется хотя бы один цикл.

Для распознавания тупика периодически производится построение графа ожидания транзакций (как уже отмечалось, иногда граф ожидания поддерживается постоянно), и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения циклов в ориентированном графе является редукция графа.

Редукция состоит в том, что прежде всего из графа ожидания удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это как бы соответствует той ситуации, что транзакции, не ожидающие удовлетворения захватов, успешно завершились и освободили захваты.) Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация исходящих дуг изменяется на противоположную (это моделирует удовлетворение захватов). После этого снова срабатывает первый шаг и так до тех пор, пока на первом шаге не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл.

Предположим, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь? Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, т.е. транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Критерием выбора является стоимость транзакции; жертвой выбирается самая дешевая транзакция. Стоимость транзакции определяется на основе многофакторной оценки, в которую с разными весами входят время выполнения, число накопленных захватов, приоритет.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный характер. При этом, естественно, освобождаются захваты и может быть продолжено выполнение других транзакций.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать.

7.3.5 Метод временных меток

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток.

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T1 выполняет следующие действия:

- Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.
- Если транзакция T не завершилась, то T1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
- Если операции T1 и T конфликтуют, то если t(T) > t(T1) (т.е. транзакция T является более «молодой», чем T), производится откат T и T1 продолжает работу.
- Если же t(T) < t(T1) (T «старше» T1), то T1 получает новую временную метку и начинается заново.

К недостаткам метода временных меток относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.

7.4 Журнализация изменений БД

Для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

- результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных;
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных.

Это, собственно, и означает, что восстанавливается последнее по времени согласованное состояние базы данных.

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных:

Индивидуальный откат транзакции. Тривиальной ситуацией отката транзакции является ее явное завершение оператором ROLLBACK. Возможны также ситуации, когда откат транзакции инициируется системой. Примерами могут быть возникновение исключительной ситуации в прикладной программе (например, деление на ноль) или выбор транзакции в качестве жертвы при обнаружении синхронизационного тупика. Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации базы данных, которые выполнялись в этой транзакции.

Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть при аварийном выключении электрического питания, при возникновении неустранимого сбоя процессора (например, срабатывании контроля оперативной памяти) и т.д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти.

Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но тем не менее, СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и журнал изменений базы данных.

Во всех трех случаях основой восстановления является избыточное хранение данных в журнале, содержащем последовательность записей об изменении базы данных.

Во всех случаях придерживаются 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 из-

мененный 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.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный журнал изменений базы данных этой транзакцией. Эти локальные журналы используются для индивидуальных откатов транзакций и могут поддерживаться в оперативной (правильнее сказать, в виртуальной) памяти. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

Этот подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант – поддержание только общего журнала изменений базы данных, который используется и при выполнении индивидуальных откатов. Далее мы рассматриваем именно этот вариант.

7.4.1 Журнализация и буферизация

Журнализация изменений тесно связана не только с управлением транзакциями, но и с буферизацией страниц базы данных в оперативной памяти. По причинам объективно существующей разницы в скорости работы процессоров и оперативной памяти и устройств внешней памяти (эта разница в скорости существовала, существует и будет существовать всегда) буферизация страниц базы данных в оперативной памяти – единственный реальный способ достижения удовлетворительной эффективности СУБД.

Если бы запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции модификации базы данных, реально немедленно записывалась бы во внешнюю память, это привело бы к существенному замедлению работы системы. Поэтому записи в журнал тоже буферизуются: при нормальной работе очередная страница выталкивается во внешнюю память журнала только при полном заполнении записями.

Но реальная ситуация является более сложной. Имеются два вида буферов – буфер журнала и буфер страниц оперативной памяти, которые содержат связанную информацию. И те, и другие буфера могут выталкиваться во внешнюю память. Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможности восстановления состояния базы данных после сбоев.

Проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое оперативной памяти не утрачено и

можно пользоваться содержимым как буфера журнала, так и буферов страниц базы данных. Но если произошел мягкий сбой, и содержимое буферов утрачено, для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц базы данных является то, что запись об изменении объекта базы данных должна попадать во внешнюю память журнала раньше, чем измененный объект оказывается во внешней памяти базы данных (протокол WAL). Другими словами, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. если во внешней памяти журнала содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Рассмотрим теперь, как можно выполнять операции восстановления базы данных в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей в соответствии с протоколом WAL.

7.4.2 Индивидуальный откат транзакции

Для того чтобы можно было выполнить по общему журналу индивидуальный откат транзакции, все записи в журнале от данной транзакции связываются в обратный список. Началом списка для незакончившихся транзакций является запись о последнем изменении базы данных, произведенном данной транзакцией. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала. Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, чтобы можно было восстановить прямой список записей об изменениях базы данных этой транзакцией.

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для незакончившихся транзакций) выполняется следующим образом:

- Выбирается очередная запись из списка данной транзакции.
- Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT, и вместо прямой операции UPDATE обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.

Любая из этих обратных операций также журнализуется. Собственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.

При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

7.4.3 Восстановление после мягкого сбоя

К числу основных проблем восстановление после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, страницу данных и несколько страниц индексов. Страницы базы данных буферизуются в оперативной памяти и выталкиваются независимо. Несмотря на применение протокола WAL, после мягкого сбоя набор страниц внешней памяти базы данных может оказаться несогласованным, т.е. часть страниц внешней памяти соответствует объекту до изменения, часть – после изменения. К такому состоянию объекта не применимы операции логического уровня.

Состояние внешней памяти базы данных называется физически согласованным, если наборы страниц всех объектов согласованы, т.е. соответствуют состоянию объекта либо после его изменения, либо до изменения.

Будем считать, что в журнале отмечаются точки физической согласованности базы данных – моменты времени, в которых во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и отсутствуют результаты операций, которые не завершились, а буфер журнала вытолкнут во внешнюю память. Назовем такие точки tpc (time of physical consistency).

Пусть к моменту мягкого сбоя состояния транзакций соответствуют диаграмме, представленной на рисунке 7.2.

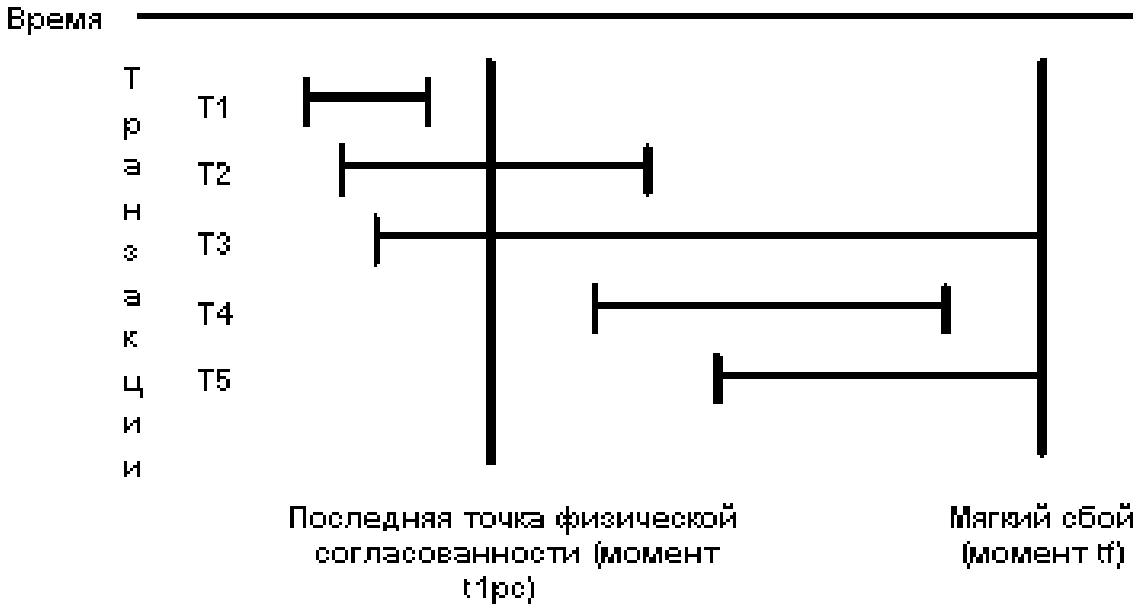


Рис. 7.2. Диаграмма состояния транзакций

Предположим, что некоторым способом удалось восстановить внешнюю память базы данных к состоянию на момент времени tlpc. Тогда:

- Для транзакции T1 никаких действий производить не требуется. Она закончилась до момента tlpc, и все ее результаты отражены во внешней памяти базы данных.
- Для транзакции T2 нужно повторно выполнить оставшуюся часть операций (redo). Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T2 после момента tlpc. Следовательно, повторная прямая интерпретация операций T2 корректна и приведет к логически согласованному состоянию базы данных (поскольку транзакция T2 успешно завершилась до момента мягкого сбоя, в журнале содержатся записи обо всех изменениях, произведенных этой транзакцией).
- Для транзакции T3 нужно выполнить в обратном направлении первую часть операций (undo). Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T3, которые были выполнены после момента tlpc. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T3, которые были выполнены до момента tlpc. Следовательно, обратная интерпретация операций T3 корректна и приведет к согласованному состоянию базы данных (поскольку транзакция T3 не завершилась к моменту мягкого сбоя, при восстановлении необходимо устранить все последствия ее выполнения).

– Для транзакции T4, которая успела начаться после момента tlpc и закончиться до момента мягкого сбоя, нужно выполнить полную повторную прямую интерпретацию операций (redo).
– Наконец, для начавшейся после момента tlpc и не успевшей завершиться к моменту мягкого сбоя транзакции T5 никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

7.4.4 Физическая согласованность базы данных

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, т.е. как восстановить состояние базы данных в момент tpc? Для этого используются два основных подхода: подход, основанный на использовании теневого механизма, и подход, в котором применяется журнализация постраничных изменений базы данных.

При открытии файла таблица отображения номеров его логических блоков в адреса физических блоков внешней памяти считывается в оперативную память. При модификации любого блока файла во внешней памяти выделяется новый блок. При этом текущая таблица отображения (в оперативной памяти) изменяется, а теневая – сохраняется неизменной. Если во время работы с открытым файлом происходит сбой, во внешней памяти автоматически сохраняется состояние файла до его открытия. Для явного восстановления файла достаточно повторно считать в оперативную память теневую таблицу отображения.

Общая идея теневого механизма показана на рисунке 7.3.

В контексте базы данных теневой механизм используется следующим образом. Периодически выполняются операции установления точки физической согласованности базы данных (checkpoints). Для этого все логические операции завершаются, все буфера оперативной памяти, содержимое которых не соответствует содержимому соответствующих страниц внешней памяти, выталкиваются. Теневая таблица отображения файлов базы данных заменяется на текущую (правильнее сказать, текущая таблица отображения записывается на место теневой).

Восстановление к tlpc происходит мгновенно: текущая таблица отображения заменяется на теневую (при восстановлении просто считывается теневая таблица отображения). Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных. Теневой механизм – это надежное, но слишком грубое средство. Обеспечивается согласованное состояние внешней памяти в один общий для всех объектов момент времени. На самом деле, достаточно иметь набор согласованных наборов страниц, каждому из которых может соответствовать свой набор времени.

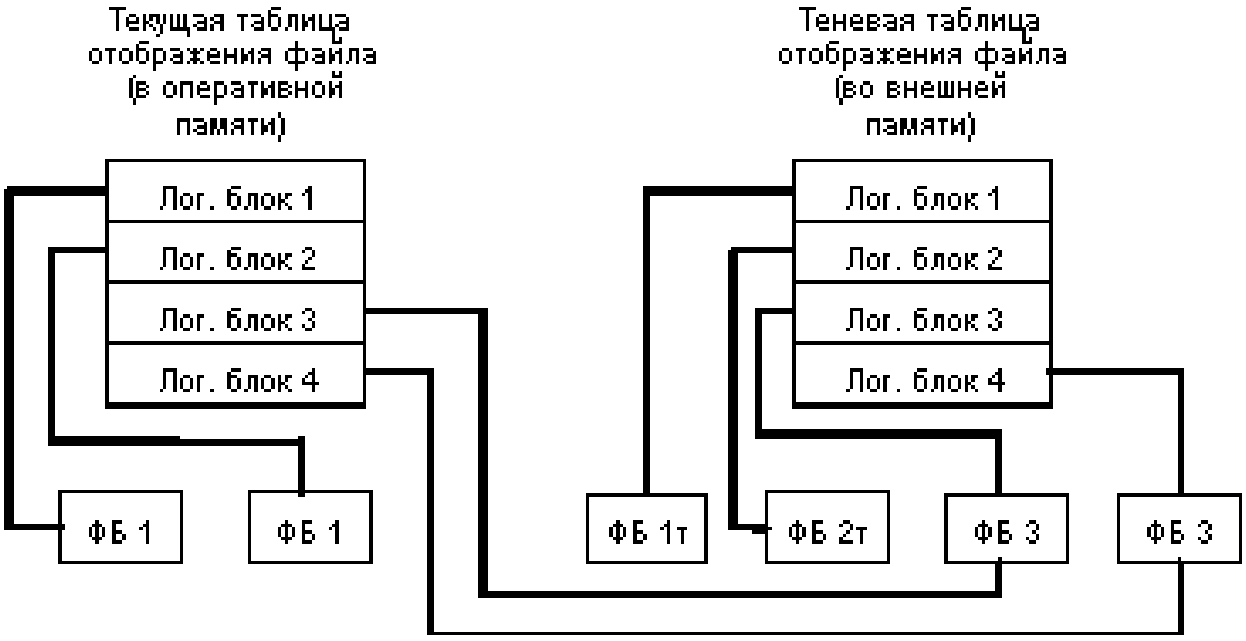


Рис. 7.3. Схема теневого механизма восстановления БД

Для достижения такого более слабого требования наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате незакончившихся логических операций. Подобно тому, как это делается с логическими записями по отношению к транзакциям, последней записью о постраничных изменениях от одной логической операции является запись о конце операции. Для того чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, при выталкивании любой страницы из буфера оперативную память в нее помещается идентификатор последней записи о постраничном изменении этой страницы.

7.4.5 Восстановление после жесткого сбоя

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя журнала изменений базы данных явно недостаточно. Основой восстановления в этом случае являются журнал и архивная копия базы данных.

Восстановление начинается с обратного копирования базы данных из архивной копии. Затем для всех закончившихся транзакций выполняется redo, т.е. операции повторно выполняются в прямом смысле.

Более точно происходит следующее:

– по журналу в прямом направлении выполняются все операции;

– для транзакций, которые не закончились к моменту сбоя, выполняется откат.

На самом деле, поскольку жесткий сбой не сопровождается утратой буферов оперативной памяти, можно восстановить базу данных до такого уровня, чтобы можно было продолжить даже выполнение незакончившихся транзакций. Но обычно это не делается, потому что восстановление после жесткого сбоя – это достаточно длительный процесс.

Хотя к ведению журнала предъявляются особые требования по части надежности, в принципе возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к архивной копии. Конечно, в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который мы рассмотрим, относится к производству архивных копий базы данных. Самый простой способ – архивировать базу данных при переполнении журнала. В журнале вводится так называемая «желтая зона», при достижении которой образование новых транзакций временно блокируется. Когда все транзакции закончатся, и следовательно, база данных придет в согласованное состояние, можно производить ее архивацию, после чего начинать заполнять журнал заново.

Можно выполнять архивацию базы данных реже, чем переполняется журнал. При переполнении журнала и окончании всех начатых транзакций можно архивировать сам журнал. Поскольку такой архивированный журнал, по сути дела, требуется только для воссоздания архивной копии базы данных, журнальная информация при архивации может быть существенно сжата.

8 ЯЗЫК ЗАПРОСОВ SQL

Прежде всего, язык SQL сочетает средства DDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД – 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. Ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при компиляции 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 видимость 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 набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или их части другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

8.1 Типы данных

В языке SQL поддерживаются следующие типы данных: CHARACTER, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION. Эти типы данных классифицируются на типы строк символов, точных чисел и приблизительных чисел.

К первому классу относится CHARACTER. Спецификатор типа имеет вид CHARACTER (lenght), где lenght задает длину строк данного типа. Заметим, что в SQL/89 нет типа строк переменного размера, хотя во мно-

гих реализациях они допускаются. Литеральные строки символов изображаются в виде 'последовательность символов' (например, 'example').

Представителями второго класса типов являются NUMERIC, DECIMAL (или DEC), INTEGER (или INT) и SMALLINT. Спецификатор типа NUMERIC имеет вид NUMERIC [(precision [, scale]). Специфицируются точные числа, представляемые с точностью precision и масштабом scale. Здесь и далее, если опущен масштаб, то он полагается равным 0, а если опущена точность, то ее значение по умолчанию определяется в реализации.

Спецификатор типа DECIMAL (или DEC) имеет вид NUMERIC [(precision [, scale]). Специфицируются точные числа, представленные с масштабом scale и точностью, равной или большей значения precision.

INTEGER специфицирует тип данных точных чисел с масштабом 0 и определяемой в реализации точностью. SMALLINT специфицирует тип данных точных чисел с масштабом 0 и определяемой в реализации точностью, не большей, чем точность чисел типа INTEGER.

Заметим еще, что в большинстве реализаций SQL поддерживаются некоторые дополнительные типы данных, например, DATE, TIME, INTERVAL, MONEY. Некоторые из этих типов специфицированы в стандарте SQL, но в текущих реализациях синтаксические и семантические свойства таких типов могут различаться.

8.2 Язык определения данных DDL

В соответствии с правилами SQL каждая таблица БД имеет *простое* и *квалифицированное* имена. В качестве квалификатора имени выступает «идентификатор полномочий» таблицы, который обычно в реализациях совпадает с именем пользователя, и квалифицированное имя таблицы имеет вид:

```
<идентификатор полномочий>.<простое имя>
```

Подход к определению схемы в SQL состоит в том, что все таблицы с одним идентификатором полномочий создаются (определяются) путем выполнения одного оператора определения схемы.

8.2.1 Оператор определения схемы

В операторе определения схемы содержится идентификатор полномочий и список элементов схемы, каждый из которых может быть определением таблицы, определением представления (view) или определением привилегий. Каждое из этих определений представляется отдельным оператором SQL, но все они должны быть встроены в оператор определения схемы.

Для этих операторов мы приведем синтаксис, поскольку это позволит более четко описать их особенности.

8.2.2 Определение таблицы

Оператор определения таблицы имеет следующий синтаксис:

```
<table definition> ::=
   CREATE TABLE  <table name>   (<table element>
   [{,<table element>}...])
<table element> ::=
  <column definition>
| <table constraint definition>
```

Кроме имени таблицы, в операторе специфицируется список элементов таблицы, каждый из которых служит либо для определения столбца, либо для определения ограничения целостности определяемой таблицы. Требуется наличие хотя бы одного определения столбца. Оператор CREATE TABLE определяет так называемую базовую таблицу, т.е. реальное хранилище данных.

Для определения столбцов таблицы и ограничений целостности используются специальные операторы, которые должны быть вложены в оператор определения таблицы.

8.2.3 Определение столбца

Оператор определения столбца описывается следующими синтаксическими правилами:

```
<column definition> ::=
    <column name> <data type>
    [<default clause>] [<column constraint>...]
<default clause> ::=
    DEFAULT { <literal> | USER | NULL }
<column constraint> ::=
    NOT NULL [<unique specification>]
  | <references specification>
  | CHECK (<search condition>)
```

Кроме обязательной части, в которой определяется имя столбца и его тип данных, определение столбца может содержать два необязательных раздела: раздел значения столбца по умолчанию и раздел ограничений целостности столбца.

В разделе значения по умолчанию указывается значение, которое должно быть помещено с строку, заносимую в данную таблицу, если зна-

чение данного столбца явно не указано. Значение по умолчанию может быть указано в виде литеральной константы с типом, соответствующим типу столбца; путем задания ключевого слова USER, которому при выполнении оператора занесения строки соответствует символьная строка, содержащая имя текущего пользователя (в этом случае столбец должен иметь тип символьных строк); или путем задания ключевого слова NULL, означающего, что значением по умолчанию является неопределенное значение. Если значение столбца по умолчанию не специфицировано и в разделе ограничений целостности столбца указано NOT NULL, то попытка занести в таблицу строку с неспецифицированным значением данного столбца приведет к ошибке.

Указание в разделе ограничений целостности NOT NULL приводит к неявному порождению проверочного ограничения целостности для всей таблицы «CHECK (C IS NOT NULL)» (где C – имя данного столбца). Если ограничение NOT NULL не указано и раздел умолчаний отсутствует, то неявно порождается раздел умолчаний DEFAULT NULL. Если указана спецификация уникальности, то порождается соответствующая спецификация уникальности для таблицы.

Если в разделе ограничений целостности указано ограничение по ссылкам данного столбца (<reference specification>), то порождается соответствующее определение ограничения по ссылкам для таблицы:

```
FOREIGN KEY(C) <reference specification>.
```

Наконец, если указано проверочное ограничение столбца, то условие поиска этого ограничения должно ссылаться только на данный столбец, и неявно порождается соответствующее проверочное ограничение для всей таблицы.

8.3 Язык запросов DQL

Ниже приведены синтаксические правила формирования запросов в стандарте SQL.

```
<cursor specification> ::=
   <query expression> [<order by clause>]
<query expression> ::=
   <query term>
|  <query expression> UNION [ALL] <query term>
<query term> ::=
   <query specification>
|  (<query expression>)
<query specification> ::=
```

```
   (SELECT [ALL | DISTINCT] <select list> <table ex-
pression>)
<select statement> ::=
   SELECT [ALL | DISTINCT] <select list>
   INTO <select target list> <table expression>
<subquery> ::=
   (SELECT [ALL | DISTINCT] <result specification>
   <table expression>
<table expression> ::=
   <from clause>
  [<where clause>]
  [<group by clause>]
  [<having clause>]
```

Язык допускает три типа синтаксических конструкций, начинающихся с ключевого слова SELECT: *спецификация курсора* (cursor specification), *оператор выборки* (select statement) и *подзапрос* (subquery). Основой для них является синтаксическая конструкция *табличное выражение* (table expression). Семантика табличного выражения состоит в том, что на основе последовательного применения разделов from, where, group by и having из заданных в разделе from таблиц строится некоторая новая результирующая таблица, порядок следования строк которой не определен и среди строк которой могут находиться дубликаты (т.е. в общем случае таблица-результат табличного выражения не является множеством). На самом деле именно структура табличного выражения наибольшим образом характеризует структуру запросов языка SQL.

8.3.1 Спецификация курсора

Наиболее общей является конструкция спецификация курсора. *Курсор* – это понятие языка SQL, позволяющее с помощью набора специальных операторов получить построчный доступ к результату запроса к БД. К табличным выражениям, участвующим в спецификации курсора, не предъявляются какие-либо ограничения. Как видно из сводки синтаксических правил, при определении спецификации курсора используются три дополнительных конструкции: спецификация запроса, выражение запросов и раздел ORDER BY.

В *спецификации запроса* задается список выборки. В общем случае это список арифметических выражений над значениями столбцов результата табличного выражения и констант, однако чаще всего это просто список столбцов. В результате применения списка выборки к результату табличного выражения производится построение новой таблицы, содержащей то же число строк, но другое число столбцов, содержащих результаты вычисления соответствующих арифметических выражений из списка выбор-

ки. В типичной ситуации – это результат выполнения операции проекции результата табличного выражения на список столбцов. Кроме того, в спецификации запроса могут содержаться ключевые слова ALL или DISTINCT. При наличии ключевого слова DISTINCT из таблицы, полученной применением списка выборки к результату табличного выражения, удаляются строки-дубликаты; при указании ALL (или просто при отсутствии DISTINCT) удаление строк-дубликатов не производится.

Выражение запросов – это выражение, строящееся по указанным синтаксическим правилам на основе спецификаций запросов. Единственной операцией, которую разрешается использовать в выражениях запросов, является операция UNION (объединение таблиц) с возможной разновидностью UNION ALL. К таблицам-операндам выражения запросов предъявляется требование, что все они должны содержать одно и то же число столбцов и соответствующие столбцы всех операндов должны быть одного и того же типа. Выражение запросов вычисляется слева направо с учетом скобок. При выполнении операции UNION производится обычное теоретико-множественное объединение операндов, т.е. из результирующей таблицы удаляются дубликаты. При выполнении операции UNION ALL образуется результирующая таблица, в которой могут содержаться строки-дубликаты.

Наконец, раздел ORDER BY позволяет установить желаемый порядок просмотра результата выражения запросов. Синтаксис ORDER BY следующий:

```
<order by clause> ::=
   ORDER BY <sort specification>
  [{,<sort specification>}...]
<sort specification> ::=
   {<unsigned integer> | <column specification>}
  [ASC | DESC]
```

Как видно из этих синтаксических правил, фактически задается список столбцов результата выражения запросов, и для каждого столбца указывается порядок просмотра строк результата в зависимости от значений этого столбца (ASC – по возрастанию (умолчание), DESC – по убыванию). Столбцы можно задавать их именами в том и только в том случае, когда (1) выражение запросов не содержит операций UNION или UNION ALL и (2) в списке выборки спецификации запроса этому столбцу соответствует арифметическое выражение, состоящее только из имени столбца. Во всех остальных случаях в разделе ORDER BY должен указываться порядковый номер столбца в таблице-результате выражения запросов.

8.3.2 Подзапрос

Подзапрос – это запрос, который может входить в предикат условия выборки оператора SQL. Поскольку подзапрос всегда вложен в некоторый другой оператор SQL, то в качестве констант в арифметическом выражении выборки и логических выражениях разделов WHERE и HAVING можно использовать значения столбцов текущих строк таблиц, участвующих в (под)запросах более внешнего уровня.

8.4 Табличное выражение

Стандарт SQL рекомендует рассматривать вычисление табличного выражения как последовательное применение разделов FROM, WHERE, GROUP BY и HAVING к таблицам, заданным в списке FROM.

8.4.1 Раздел FROM

Раздел FROM имеет следующий синтаксис:

```
<from clause> ::=
   FROM <table reference>
 ({,<table reference>}...]
<table reference> ::=
   <table name> [<correlation name>]
```

Результатом выполнения раздела FROM является расширенное декартово произведение таблиц, заданных списком таблиц раздела FROM.

Как видно из синтаксиса, рядом с именем таблицы можно указывать еще одно имя «correlation name». Фактически это некоторый синоним (alias) имени таблицы, который можно использовать в других разделах табличного выражения для ссылки на строки именно этого вхождения таблицы.

Если табличное выражение содержит только раздел FROM (это единственный обязательный раздел табличного выражения), то результат табличного выражения совпадает с результатом раздела FROM.

Примеры.

Декартово произведение отношений R1 и R2:

```
SELECT * FROM R1, R2
```

Проекция декартова произведения R1⊕R2 на столбцы A и B:

```
SELECT R1.A, R2.B FROM R1, R2
```

8.4.2 Предложение WHERE

Если в табличном выражении присутствует предложение WHERE, то оно вычисляется следующим за FROM. Синтаксис предложения WHERE следующий:

```
<where clause> ::= WHERE <search condition>
<search condition> ::=
   <boolean term>
 ( <search condition> OR <boolean term>
<Boolean term> ::=
   <boolean factor>
 ( <boolean term> AND <boolean factor>
<boolean factor> ::= [NOT] <boolean primary>
<boolean primary> ::= <predicate> | (<search condi-
tion>)
```

Вычисление предложения WHERE производится по следующим правилам: Пусть R – результат вычисления раздела FROM. Тогда условие (search condition) применяется ко всем строкам R, и результатом предложения WHERE является таблица, состоящая из тех строк R, для которого результатом вычисления условия является true. Если условие выборки включает подзапросы, то каждый подзапрос вычисляется для каждого кортежа таблицы R.

Заметим, что поскольку SQL допускает наличие в базе данных неопределенных значений, то вычисление условия поиска производится не в булевой, а в трехзначной логике со значениями true, false и null (неизвестно). Для любого предиката известно, в каких ситуациях он может порождать значение unknown. Булевские операции AND, OR и NOT работают в трехзначной логике следующим образом:

```
true AND null = null
false AND null = false
null AND null = null
true OR null = true
false OR null = null
null OR null = null
NOT null = null
```

Среди предикатов условия поиска в соответствии с SQL могут находиться следующие предикаты: предикат сравнения, предикат between, предикат in, предикат like, предикат null и предикаты exists, any, all. Сразу заметим, что во всех реализациях SQL на эффективность выполнения запро-

са существенно влияет наличие в условии поиска простых предикатов сравнения (предикатов, задающих сравнение столбца таблицы с константой). Наличие таких предикатов позволяет СУБД использовать индексы при выполнении запроса, т.е. избегать полного просмотра таблицы.

Синтаксис *предиката сравнения* определяется следующими правилами:

```
<comparison predicate> ::=
   <value expression> <comp op>
  {<value expression> | <subquery>}
<comp op> ::=
   = | <> | < | > | <= | >=
```

Арифметические выражения левой и правой частей предиката сравнения строятся по общим правилам построения арифметических выражений и могут включать в общем случае имена столбцов таблиц из раздела FROM и константы. Типы данных арифметических выражений должны быть сравнимыми (например, если тип столбца a таблицы A является типом символьных строк, то предикат «a = 5» недопустим).

Если правый операнд операции сравнения задается подзапросом, то дополнительным ограничением является то, что мощность результата подзапроса должна быть не более единицы. Если хотя бы один из операндов операции сравнения имеет неопределенное значение, или если правый операнд является подзапросом с пустым результатом, то значение предиката сравнения равно null.

Заметим, что значение арифметического выражения не определено, если в его вычислении участвует хотя бы одно неопределенное значение. Еще одно важное замечание: в контексте GROUP BY, DISTINCT и ORDER BY неопределенное значение выступает как специальный вид определенного значения, т.е. возможно, например, образование группы строк, значение указанного столбца которых является неопределенным.

Предикат between имеет следующий синтаксис:

```
<between predicate> ::=
   <value expression>
   [NOT] BETWEEN <value expression> AND <value ex-
pression>
```

Результат «x BETWEEN y AND z» тот же самый, что результат «x >= y AND x <= z». Результат «x NOT BETWEEN y AND z» тот же самый, что результат «NOT (x BETWEEN y AND z)».

Предикат in определяется следующими синтаксическими правилами:

```
<in predicate> ::=
   <value expression> [NOT] IN
  {<subquery> | (<in value list>)}
<in value list> ::=
   <value specification>
 {,<value specification>}...
```

Типы левого операнда и значений из списка правого операнда (напомним, что результирующая таблица подзапроса должна содержать ровно один столбец) должны быть сравнимыми.

Значение предиката равно true в том и только в том случае, когда значение левого операнда совпадает хотя бы с одним значением списка правого операнда. Если список правого операнда пуст (так может быть, если правый операнд задается подзапросом) или значение «подразумеваемого» предиката сравнения x = y (где x – значение арифметического выражения левого операнда) равно false для каждого элемента y списка правого операнда, то значение предиката in равно false. В противном случае значение предиката in равно null. По определению значение предиката «x NOT IN S» равно значению предиката «NOT (x IN S)».

Предикат like имеет следующий синтаксис:

```
<like predicate> ::=
   <column specification> [NOT] LIKE <pattern>
   [ESCAPE <escape character>]
<pattern> ::= <value specification>
   <escape character> ::= <value specification>
```

Типы данных столбца левого операнда и образца должны быть типами символьных строк. В разделе ESCAPE должен специфицироваться одиночный символ.

Значение предиката равно true, если pattern является подстрокой заданного столбца. При этом если раздел ESCAPE отсутствует, то при сопоставлении шаблона со строкой производится специальная интерпретация двух символов шаблона: символ подчеркивания («_») обозначает любой одиночный символ; символ процента («%») обозначает последовательность произвольных символов произвольной длины (может быть, нулевой).

Если же раздел ESCAPE присутствует и специфицирует некоторый одиночный символ x, то пары символов «x_» и «x%» представляют одиночные символы «_» и «%» соответственно.

Значение предиката like есть unknown, если значение столбца, либо шаблона не определено.

Значение предиката «x NOT LIKE y ESCAPE z» совпадает со значением «NOT x LIKE y ESCAPE z».

Предикат null описывается синтаксическим правилом:

```
<null predicate> ::=
   <column specification> IS [NOT] NULL
```

Этот предикат всегда принимает значения true или false. При этом значение «x IS NULL» равно true тогда и только тогда, когда значение x не определено. Значение предиката «x NOT IS NULL» равно значению «NOT x IS NULL».

Предикат exists имеет следующий синтаксис:

```
<exists predicate> ::=
   EXISTS <subquery>
```

Значением этого предиката всегда является true или false, и это значение равно true тогда и только тогда, когда результат вычисления подзапроса не пуст.

Предикат any имеет следующий синтаксис:

```
<any predicate> ::=
   <value expression> <comp op> ANY <subquery>
```

Значением этого предиката всегда является true или false, и это значение равно true тогда и только тогда, когда предикат сравнения справедлив хотя бы для одного кортежа результата подзапроса.

Предикат all имеет следующий синтаксис:

```
<all predicate> ::=
   <value expression> <comp op> ALL <subquery>
```

Значением этого предиката всегда является true или false, и это значение равно true тогда и только тогда, когда предикат сравнения справедлив для всех кортежей результата подзапроса.

Примеры. Пусть база данных, предназначенная для учета результатов сессии, состоит из следующих отношений:

- Сводная ведомость – `Ведом(ФИО, Дисц, Оценка);`
- Список студенческих групп – `Группы(ФИО, Группа);`
- Список дисциплин – `Дисц(Группа, Дисц).`

1) Выдать список дисциплин, по которым проводится контроль знаний в текущую сессию:

```
SELECT DISTINCT Дисц FROM Дисц
```

Префикс (имя отношения) можно опустить, если это не приводит к неоднозначной трактовке списка выборки.

2) Выдать список студентов, сдавших физику на «отлично»:

```
SELECT ФИО FROM Ведом
      WHERE Дисц = "Физика" AND Оценка = 5
```

3) Выдать список студентов, получивших несколько неудовлетворительных оценок на экзаменах:

```
SELECT DISTINCT ФИО FROM Ведом A, Ведом B
      WHERE A.ФИО = B.ФИО AND A.Дисц <> B.Дисц
            AND A.Оценка = 2 AND B.Оценка = 2
```

В данном примере запрос обрабатывает два экземпляра отношения Ведом, для которых введены псевдонимы A и B.

4) Выдать список студентов, не явившихся на экзамены, и дисциплины:

```
SELECT ФИО, Дисц FROM Ведом WHERE Оценка IS NULL
```

8.4.3 Предложение GROUP BY

Если в табличном выражении присутствует предложение GROUP BY, то оно выполняется следом за WHERE. Синтаксис раздела GROUP BY следующий:

```
<group by clause> ::=
   GROUP BY <column specification>
   [{,<column specification>}...]
```

Если обозначить через R таблицу, являющуюся результатом предыдущего раздела (FROM или WHERE), то результатом предложения GROUP BY является разбиение R на множество групп строк, состоящего из минимального числа групп, таких, что для каждого столбца из списка столбцов раздела GROUP BY во всех строках каждой группы, включающей более одной строки, значения этого столбца равны.

8.4.4 Предложение HAVING

Наконец, последним при вычислении табличного выражения используется предложение HAVING (если оно присутствует). Синтаксис этого предложения следующий:

```
<having clause> ::=
   HAVING <search condition>
```

Предложение HAVING может осмысленно появиться в табличном выражении только в том случае, когда в нем присутствует предложение GROUP BY. Условие этого раздела задается для групп строк сгруппированной таблицы. Формально предложение HAVING может присутствовать и в табличном выражении, не содержащем GROUP BY. В этом случае полагается, что результат вычисления предыдущих разделов представляет собой сгруппированную таблицу, состоящую из одной группы без выделенных столбцов группирования.

Условие отбора групп предложения HAVING строится по тем же синтаксическим правилам, что и условие отбора кортежей предложения WHERE, и может включать те же самые предикаты. Однако имеются специальные синтаксические ограничения по части использования в условии поиска спецификаций столбцов таблиц из раздела FROM данного табличного выражения. Эти ограничения следуют из того, что условие раздела HAVING задает условие на целую группу, а не на индивидуальные кортежи.

Поэтому в арифметических выражениях предикатов, входящих в условие выборки предложения HAVING, прямо можно использовать только спецификации столбцов, указанных в качестве столбцов группирования в разделе GROUP BY. Остальные столбцы можно специфицировать только внутри спецификаций агрегатных функций COUNT, SUM, AVR, MIN и MAX, вычисляющих в данном случае некоторое агрегатное значение для всей группы строк. Аналогично обстоит дело с подзапросами, входящими в предикаты условия выборки предложения HAVING: если в подзапросе используется характеристика текущей группы, то она может задаваться только путем ссылки на столбцы группирования.

Результатом выполнения предложения HAVING является сгруппированная таблица, содержащая только те группы строк, для которых результат вычисления условия поиска есть true. В частности, если предложение HAVING присутствует в табличном выражении, не содержащем GROUP BY, то результатом его выполнения будет либо пустая таблица, либо результат выполнения предыдущих разделов табличного выражения, рассматриваемый как одна группа без столбцов группирования.

8.5 Агрегатные функции и результаты запросов

Агрегатные функции (в стандарте SQL они называются функциями над множествами) определяются в SQL следующими синтаксическими правилами:

```
<set function specification> ::=
   COUNT(*) | <distinct set function>
            | <all set function>
<distinct set function> ::=
   { AVG | MAX | MIN | SUM | COUNT }
    (DISTNICT <column specification>)
<all set function> ::=
   { AVR | MAX | MIN | SUM } ([ALL] <value expres-
sion>)
```

Как видно из этих правил, в стандарте SQL определены пять агрегатных функций:

- COUNT – число строк или значений;
- MAX – максимальное значение;
- MIN – минимальное значение;
- SUM – суммарное значение;
- AVR – среднее значение.

8.5.1 Семантика агрегатных функций

Агрегатные функции предназначены для того, чтобы вычислять некоторое значение для заданного множества строк. Таким множеством может быть группа строк, если агрегатная функция применяется к сгруппированной таблице, или вся таблица. Для всех агрегатных функций, кроме COUNT(*), фактический (т.е. требуемый семантикой) порядок вычислений следующий: на основании параметров агрегатной функции из заданного множества строк производится список значений. Затем по этому списку значений производится вычисление функции. Если список оказался пустым, то значение функции COUNT для него есть 0, а значение всех остальных функций – null.

Пусть T обозначает тип значений из этого списка. Тогда результат вычисления функции COUNT – точное число с масштабом и точностью, определяемыми в реализации. Тип результата значений функций MAX и MIN совпадает с T. При вычислении функций SUM и AVR тип T не должен быть типом символьных строк, а тип результата функции – это тип точных чисел с определяемыми в реализации масштабом и точностью, если T – тип точных чисел, и тип приблизительных чисел с определяемой в реализации точностью, если T – тип приблизительных чисел.

Вычисление функции COUNT(*) производится путем подсчета числа строк в заданном множестве. Все строки считаются различными, даже если они состоят из одного столбца со значением null во всех строках.

Если агрегатная функция специфицирована с ключевым словом DISTINCT, то список значений строится из значений указанного столбца. (Подчеркнем, что в этом случае не допускается вычисление арифметических выражений!) Далее из этого списка удаляются неопределенные значения, и в нем устраняются значения-дубликаты. Затем вычисляется указанная функция.

Если агрегатная функция специфицирована без ключевого слова DISTINCT (или с ключевым словом ALL), то список значений формируется из значений арифметического выражения, вычисляемого для каждой строки заданного множества. Далее из списка удаляются неопределенные значения, и производится вычисление агрегатной функции. Обратите внимание, что в этом случае не допускается применение функции COUNT!

8.5.2 Результаты запросов

Агрегатные функции можно разумно использовать в спецификации курсора, операторе выборки и подзапросе после ключевого слова SELECT (будем называть в этом подразделе все такие конструкции списком выборки), и в условии выборки раздела HAVING. Стандарт допускает более экзотические использования агрегатных функций в подзапросах (агрегатная функция на группе кортежей внешнего запроса), но на практике они встречаются очень редко.

Рассмотрим различные случаи применения агрегатных функций в списке выборки в зависимости от вида табличного выражения.

Если результат табличного выражения R не является сгруппированной таблицей, то появление хотя бы одной агрегатной функции от множества строк R в списке выборки приводит к тому, что R неявно рассматривается как сгруппированная таблица, состоящая из одной (или нуля) групп с отсутствующими столбцами группирования. Поэтому в этом случае в списке выборки не допускается прямое использование спецификаций строк R: все они должны находиться внутри спецификаций агрегатных функций. Результатом запроса является таблица, состоящая не более чем из одной строки, полученной путем применения агрегатных функций к R.

Аналогично обстоит дело в том случае, когда R представляет собой сгруппированную таблицу, но табличное выражение не содержит раздела GROUP BY (и, следовательно, содержит раздел HAVING). Если в предыдущем случае было два варианта формирования списка выборки: только с прямым указанием столбцов R или только с указанием их внутри спецификаций агрегатных функций, то в данном случае возможен только второй вариант. Результат табличного выражения явно объявлен сгруппированной таблицей, состоящей из одной группы, и результат запроса можно форми-

ровать только путем применения агрегатных функций к этой группе строк. Опять результатом запроса является таблица, состоящая не более чем из одной строки, полученной путем применения агрегатных функций к R.

Наконец, рассмотрим случай, когда R представляет собой «настоящую» сгруппированную таблицу, т.е. табличное выражение содержит раздел GROUP BY и, следовательно, определен, по крайней мере, один столбец группирования. В этом случае правила формирования списка выборки полностью соответствуют правилам формирования условия выборки раздела HAVING: допускает прямое использование спецификации столбцов группирования, а спецификации остальных столбцов R могут появляться только внутри спецификаций агрегатных функций. Результатом запроса является таблица, число строк в которой равно числу групп в R, и каждая строка формируется на основе значений столбцов группирования и агрегатных функций для данной группы.

Примеры.

1) По каждой из дисциплин выдать количество различных оценок в сводной ведомости:

```
SELECT Дисц, COUNT(DISTINCT Оценка) FROM Ведом
      WHERE Оценка IS NOT NULL
      GROUP BY Дисц
```

2) Для каждой из групп и дисциплин выдать количество студентов, успешно сдавших экзамены, и средний балл:

```
SELECT Группы.Группа, Ведом.Дисц, COUNT(*),
AVR(Оценка)
      FROM Группы, Ведом
      WHERE Ведом.ФИО = Группы.ФИО
            AND Ведом.Оценка IS NOT NULL
            AND Ведом.Оценка > 2
      GROUP BY Ведом.Дисц, Группы.Группа
```

3) Выдать список групп, в которых по одной и той же дисциплине получено более одного «неуда»:

```
SELECT Группы.Группа, FROM Ведом, Группы
      WHERE Ведом.ФИО = Группы.ФИО AND Ведом.Оценка = 2
      GROUP BY Ведом.Дисц, Группы.Группа
      HAVING COUNT(*) > 1
```

В этом примере агрегатная функция используется в условии отбора групп.

4) Выдать список студентов, успешно сдавших все экзамены:

```
SELECT ФИО FROM Ведом
      WHERE Оценка IS NOT NULL AND Оценка > 2
      GROUP BY ФИО
      HAVING COUNT(*) = (SELECT COUNT(*)FROM Группы, Дисц
            WHERE Группы.Группа = Дисц.Группа
                  AND Ведом.ФИО = Группы.ФИО)
```

Вложенный запрос возвращает количество дисциплин, по которым должен сдать экзамен текущий студент из внешнего запроса. COUNT во внешнем запросе подсчитывает количество успешно сданных этим студентом экзаменов.

Рассмотрим базу данных для учета поставок комплектующих на предприятие. Она состоит из двух отношений:
Ассортимент комплектующих – `Ассорт(Ном_Дет,Наименование);`
Список поставок – `Поставки(Поставщик, Ном_Дет, Дата).`

5) Выдать список поставщиков, которые поставляют весь ассортимент комплектующих:

```
SELECT DISTINCT Поставщик FROM Поставки sp
      WHERE NOT EXIST (SELECT Ном_Дет FROM Ассорт
            WHERE NOT EXIST (select * FROM Поставки sp1
                  WHERE sp1.Поставщик = sp.Поставщик
                  and sp1.Ном_Дет = Ассорт.Ном_Дет))
```

Фактически список поставщиков формируется по условию двойного отрицания: в него попадают только те поставщики, для которых в ассортименте нет такой детали, которую они не поставляли! Для иллюстрации многовариантности реализации запросов в SQL приведем второй вариант этого запроса, реализованный с помощью селекции групп:

```
SELECT DISTINCT Поставщик FROM Поставки
      GROUP BY Поставщик
      HAVING COUNT(DISTINCT Ном_Дет) =
            (SELECT COUNT(Ном_Дет) FROM Ассорт)
```

Отметим, что данный запрос реализуется с помощью операции деления отношений Поставки [Ном_Дет:Ном_Дет] Ассотр. Таким образом, здесь приведены два варианта описания операции деления отношений на SQL.

Выдать список студентов, сдавших сессию на 4 и 5:

```
SELECT ФИО FROM Ведом
      WHERE 4 >= ALL
            (SELECT Ведом.Оценка FROM Ведом w
WHERE ФИО = w.ФИО)
```

6) Предположим, что учет успеваемости на лабораторных занятиях ведется с помощью отношения: `Лаб(ФИО, Дисц, Лаб_раб, Оценка)`

Выдать список студентов, получивших на экзамене оценку не меньшую, чем хотя бы одна из оценок по лабораторным работам по данной дисциплине

```
SELECT ФИО FROM Ведом
      WHERE Ведом.Оценка >= ANY
             (SELECT Лаб.Оценка FROM Лаб
                  WHERE Ведом.Дисц = Лаб.Дисц AND
                        Ведом.ФИО = Лаб.ФИО)
```

8.6 Объединения

Объединения были введены в стандарте SQL2. Результат объединения используется разделом FROM аналогично списку таблиц.

Пусть: R = {r}; Q = {q}, схемы отношений S_R = {A}; S_Q = {B} имеют эквивалентные наборы атрибутов C. Тогда в условии объединения должны фигурировать атрибуты из набора C.

Шаблон выражения объединения выглядит следующим образом:

```
Левое_отн {INNER | {LEFT | RIGHT | FULL} [OUTER]}
JOIN Правое_отн {ON условие | USING (список столб-
цов)}
```

Различают внутренние объединения (inner), внешние (outer) – правые (right), левые (left) и полные (full).

Внутреннее объединение. Результат состоит из сцепления кортежей (r, q), для которых справедливо условие. Разновидностью внутреннего является естественное объединение natural inner, при котором из (r, q) удаляются дубликаты значений набора атрибутов C.

Левое внешнее объединение. Результат состоит из сцепления кортежей (r, q). В него входят все кортежи левого отношения R и те кортежи Q, для которых справедливо условие. Если условие не выполняется, то значения атрибутов {B} правого отношения Q устанавливаются в NULL.

Правое внешнее объединение. В результат объединения входят все кортежи правого отношения Q и те кортежи R, для которых справедливо условие. Если условие не выполняется, то значения атрибутов {A} левого отношения R устанавливаются в NULL.

Полное внешнее объединение. В результат объединения входят все кортежи правого и левого отношений. Если условие не выполняется, то значения атрибутов {A} и {B} устанавливаются в NULL поочередно.

Пример. Сформировать сводную ведомость по результатам сессии.

```
SELECT ФИО, Дисц, Оценка FROM
      (Группы NATURAL INNER JOIN Дисц) LEFT JOIN Ведом
            USING (ФИО, Дисц)
```

USING задает атрибуты, равенство значений которых является условием объединения.

8.7 Язык манипулирования данными DML

Манипулирование данными в SQL сводится к трем операциям: добавление (вставки) кортежей (INSERT), удаления (DELETE) и модификации (UPDATE).

8.7.1 Добавление кортежей

Синтаксис оператора вставки кортежа следующий:

```
<insert statement> ::=
   INSERT INTO <table name> [<column names>]
   VALUES <value expressions>
```

Вставляется один кортеж с заданными значениями атрибутов. Если список столбцов отсутствует, то значения в списке должны соответствовать доменам атрибутов в порядке их следования в схеме отношения.

Вставка нескольких кортежей:

```
<insert statement> ::=
   INSERT INTO <table name> [(<column names>)]
   <query expression>
```

Схема результата запроса должна соответствовать списку <column names>, либо, при его отсутствии, схеме отношения.

8.7.2 Удаление кортежей

Оператор удаления кортежа из отношения:

```
<delete statement > ::=
   DELETE FROM <table name>
   [WHERE <search condition>]
```

Таблица T, указанная в разделе FROM оператора DELETE, должна быть изменяемой. На условие поиска накладывается то условие, что на таблицу T не должны содержаться ссылки ни в каком вложенном подзапросе предикатов раздела WHERE.

Фактически оператор выполняется следующим образом: последовательно просматриваются все строки таблицы T, и те строки, для которых результатом вычисления условия выборки является true, удаляются из таблицы T. При отсутствии раздела WHERE удаляются все строки таблицы T.

8.7.3 Модификация кортежей

Оператор модификации обладает следующим синтаксисом:

```
<update statement> ::=
   UPDATE <table name>
   SET <set clause>
 [{,<set clause>}...]
   [WHERE <search conditions>]
<set clause> ::=
  <column name> =
{ <value expression> | NULL }
```

Таблица T, указанная в операторе UPDATE, должна быть изменяемой. На условие поиска накладывается то условие, что на таблицу T не должны содержаться ссылки ни в каком вложенном подзапросе предикатов раздела WHERE.

Оператор фактически выполняется следующим образом: таблица T последовательно просматривается, и каждая строка, для которой результатом вычисления условия поиска является true, изменяется в соответствии с разделом SET. Если арифметическое выражение в разделе SET содержит ссылки на столбцы таблицы T, то при вычислении арифметического выражения используются значения столбцов текущей строки до их модификации. При отсутствии раздела WHERE модифицируются все строки таблицы T.

Примеры.

1) Зачислить нового студента:

```
INSERT INTO Группы (ФИО, Группа)
      VALUES ("Петров И.И.", "ИС-Р41")
```

2) Добавить пустую ведомость по дисциплине «Базы данных» для группы ИС-Р41:

```
INSERT INTO Ведом (ФИО, Дисц)
      SELECT ФИО, Дисц FROM Группы, Дисц
            WHERE Группы.Группа = Дисц.Группа AND
            Дисц.Дисц = "БД" AND
            Группы.Группа = "ИС-Р41"
```

3) Отчислить задолжников, имеющих больше двух неудов:

```
DELETE FROM Группы WHERE ФИО IN
      (SELECT Ведом.ФИО FROM Ведом
            WHERE Оценка = 2
            GROUP BY Ведом.ФИО
            HAVING COUNT(*) >= 2)
```

Отметим, что в данном случае аргументом предиката IN является результат запроса.

4) Зафиксировать пересдачу экзамена по Базам данных студентом И.И. Ивановым:

```
UPDATE Ведом SET Оценка = 4
      WHERE ФИО = "Иванов И.И." AND Дисц = "БД"
```

5) Для учета студентов, получающих стипендию, необходимо добавить атрибут Стипендия в схему отношения Группы:

```
Группы(ФИО, Группа, Стипендия)
```

Начисление стипендии по результатам сессии:

```
UPDATE Группы SET Стипендия = True
      WHERE Группы.ФИО NOT IN
            (SELECT Ведом.ФИО FROM Ведом
                  WHERE Оценка IS NOT NULL
                        AND Оценка <= 3 )
```

Всем студентам, сдавшим сессию без троек, начисляется стипендия.

9 СУБД В АРХИТЕКТУРЕ «КЛИЕНТ-СЕРВЕР»

Применительно к системам баз данных архитектура «клиент-сервер» интересна и актуальна главным образом потому, что обеспечивает простое и относительно дешевое решение проблемы коллективного доступа к базам данных в локальной сети. В некотором роде системы баз данных, основанные на архитектуре «клиент-сервер», являются приближением к распределенным системам баз данных, конечно, существенно упрощенным приближением, но зато не требующим решения основного набора проблем действительно распределенных баз данных.

9.1 Открытые системы

Реальное распространение архитектуры «клиент-сервер» стало возможным благодаря развитию и широкому внедрению в практику концепции открытых систем. Поэтому мы начнем с краткого введения в открытые системы.

Основным смыслом подхода открытых систем является упрощение комплексирования вычислительных систем за счет международной и национальной стандартизации аппаратных и программных интерфейсов. Главной побудительной причиной развития концепции открытых систем явились повсеместный переход к использованию локальных компьютерных сетей и те проблемы комплексирования аппаратно-программных средств, которые вызвал этот переход. В связи с бурным развитием технологий глобальных коммуникаций открытые системы приобретают еще большее значение и масштабность.

Ключевой фразой открытых систем, направленной в сторону пользователей, является независимость от конкретного поставщика. Ориентируясь на продукцию компаний, придерживающихся стандартов открытых систем, потребитель, который приобретает любой продукт такой компании, не попадает к ней в рабство. Он может продолжить наращивание мощности своей системы путем приобретения продуктов любой другой компании, соблюдающей стандарты. Причем это касается как аппаратных, так и программных средств и не является необоснованной декларацией. Реальная возможность независимости от поставщика проверена в отечественных условиях.

Технологии и стандарты открытых систем обеспечивают реальную и проверенную практикой возможность производства системных и прикладных программных средств со свойствами *мобильности* (portability) и *интероперабельности* (interoperability). Свойство мобильности означает сравнительную простоту переноса программной системы в широком спектре аппаратно-программных средств, соответствующих стандартам. Интер-

операбельность означает упрощения комплексирования новых программных систем на основе использования готовых компонентов со стандартными интерфейсами.

Преимуществом такого подхода для пользователей является то, что они могут постепенно заменять компоненты системы на более совершенные, не утрачивая работоспособности системы. В частности, в этом кроется решение проблемы постепенного наращивания вычислительных, информационных и других мощностей компьютерной системы.

9.2 Клиенты и серверы локальных сетей

В основе широкого распространения локальных сетей компьютеров лежит известная идея разделения ресурсов. Высокая пропускная способность локальных сетей обеспечивает эффективный доступ из одного узла локальной сети к ресурсам, находящимся в других узлах.

Развитие этой идеи приводит к функциональному выделению компонентов сети: разумно иметь не только доступ к ресурсами удаленного компьютера, но также получать от этого компьютера некоторый сервис, который специфичен для ресурсов данного рода и программные средства для обеспечения которого нецелесообразно дублировать в нескольких узлах. Так мы приходим к различению рабочих станций и серверов локальной сети.

Рабочая станция предназначена для непосредственной работы пользователя или категории пользователей и обладает ресурсами, соответствующими локальным потребностям данного пользователя. Специфическими особенностями рабочей станции могут быть объем оперативной памяти (далеко не все категории пользователей нуждаются в наличии большой оперативной памяти), наличие и объем дисковой памяти (достаточно популярны бездисковые рабочие станции, использующие внешнюю память дискового сервера), характеристики процессора и монитора (некоторым пользователям нужен мощный процессор, других в большей степени интересует разрешающая способность монитора, для третьих обязательно требуются средства убыстрения графики и т.д.). При необходимости можно использовать ресурсы и/или услуги, предоставляемые сервером.

Сервер локальной сети должен обладать ресурсами, соответствующими его функциональному назначению и потребностям сети. Заметим, что в связи с ориентацией на подход открытых систем, правильнее говорить о логических серверах (имея в виду набор ресурсов и программных средств, обеспечивающих услуги над этими ресурсами), которые располагаются не обязательно на разных компьютерах. Особенностью логического сервера в открытой системе является то, что если по соображениям эффективности сервер целесообразно переместить на отдельный компьютер, то это можно проделать без потребности в какой-либо переделке как его самого, так и использующих его прикладных программ.

Примерами серверов могут служить:

– сервер телекоммуникаций, обеспечивающий услуги по связи данной локальной сети с внешним миром;
– вычислительный сервер, дающий возможность производить вычисления, которые невозможно выполнить на рабочих станциях;
– дисковый сервер, обладающий расширенными ресурсами внешней памяти и предоставляющий их в использование рабочим станциями и, возможно, другим серверам;
– файловый сервер, поддерживающий общее хранилище файлов для всех рабочих станций;
– сервер баз данных фактически обычная СУБД, принимающая запросы по локальной сети и возвращающая результаты.

Сервер локальной сети предоставляет ресурсы (услуги) рабочим станциям и/или другим серверам.

Принято называть клиентом локальной сети, запрашивающим услуги у некоторого сервера и сервером, компонент локальной сети, оказывающий услуги некоторым клиентам.

9.3 Системная архитектура «клиент-сервер»

Понятно, чтобы прикладная программа, выполняющаяся на рабочей станции, могла запросить услугу у некоторого сервера, как минимум требуется некоторый интерфейсный программный слой, поддерживающий такого рода взаимодействие (было бы, по меньшей мере, неестественно требовать, чтобы прикладная программа напрямую пользовалась примитивами транспортного уровня локальной сети). Из этого, собственно, и вытекают основные принципы системной архитектуры «клиент-сервер».

Система разбивается на две части, которые могут выполняться в разных узлах сети, – клиентскую и серверную. Прикладная программа или конечный пользователь взаимодействуют с клиентской частью системы, которая в простейшем случае обеспечивает просто надсетевой интерфейс. Клиентская часть системы при потребности обращается по сети к серверной части. Заметим, что в развитых системах сетевое обращение к серверной части может и не понадобиться, если система может предугадывать потребности пользователя, и в клиентской части содержатся данные, способные удовлетворить его следующий запрос.

Интерфейс серверной части определен и фиксирован. Поэтому возможно создание новых клиентских частей существующей системы (пример интероперабельности на системном уровне).

Основной проблемой систем, основанных на архитектуре «клиент-сервер», является то, что в соответствии с концепцией открытых систем от них требуется мобильность в как можно более широком классе аппаратно-программных решений открытых систем. Даже если ограничиться UNIX-

ориентированными локальными сетями, в разных сетях применяется разная аппаратура и протоколы связи. Попытки создания систем, поддерживающих все возможные протоколы, приводят к их перегрузке сетевыми деталями в ущерб функциональности.

Еще более сложный аспект этой проблемы связан с возможностью использования разных представлений данных в разных узлах неоднородной локальной сети. В разных компьютерах может существовать различная адресация, представление чисел, кодировка символов и т.д. Это особенно существенно для серверов высокого уровня: телекоммуникационных, вычислительных, баз данных.

Общим решением проблемы мобильности систем, основанных на архитектуре «клиент-сервер», является опора на программные пакеты, реализующие протоколы удаленного вызова процедур (RPC – Remote Procedure Call). При использовании таких средств обращение к сервису в удаленном узле выглядит как обычный вызов процедуры. Средства RPC, в которых, естественно, содержится вся информация о специфике аппаратуры локальной сети и сетевых протоколов, переводит вызов в последовательность сетевых взаимодействий. Тем самым, специфика сетевой среды и протоколов скрыта от прикладного программиста.

При вызове удаленной процедуры программы RPC производят преобразование форматов данных клиента в промежуточные машинно-независимые форматы и затем преобразование в форматы данных сервера. При передаче ответных параметров производятся аналогичные преобразования.

Если система реализована на основе стандартного пакета RPC, она может быть легко перенесена в любую открытую среду.

9.4 Серверы баз данных

Термин «сервер баз данных» обычно используют для обозначения всей СУБД, основанной на архитектуре «клиент-сервер», включая и серверную, и клиентскую части. Такие системы предназначены для хранения и обеспечения доступа к базам данных.

Хотя обычно одна база данных целиком хранится в одном узле сети и поддерживается одним сервером, серверы баз данных представляют собой простое и дешевое приближение к распределенным базам данных, поскольку общая база данных доступна для всех пользователей локальной сети.

9.4.1 Принципы взаимодействия между клиентскими и серверными частями

Доступ к базе данных от прикладной программы или пользователя производится путем обращения к клиентской части системы. В качестве основного интерфейса между клиентской и серверной частями выступает язык баз данных SQL.

Это язык по сути дела представляет собой текущий стандарт интерфейса СУБД в открытых системах. Собирательное название SQL-сервер относится ко всем серверам баз данных, основанных на SQL. Соблюдая предосторожности при программировании, некоторые из которых были рассмотрены на предыдущих лекциях, можно создавать прикладные информационные системы, мобильные в классе SQL-серверов.

Серверы баз данных, интерфейс которых основан исключительно на языке SQL, обладают своими преимуществами и недостатками. Очевидное преимущество – стандартность интерфейса. В пределе, хотя пока это не совсем так, клиентские части любой SQL-ориентированной СУБД могли бы работать с любым SQL-сервером вне зависимости от того, кто его произвел.

Недостаток тоже очевиден. При таком высоком уровне интерфейса между клиентской и серверной частями системы на стороне клиента работает слишком мало программ СУБД. Это нормально, если на стороне клиента используется маломощная рабочая станция. Но если клиентский компьютер обладает достаточной мощностью, то часто возникает желание возложить на него больше функций управления базами данных, разгрузив сервер, который является узким местом всей системы.

Одним из перспективных направлений СУБД является гибкое конфигурирование системы, при котором распределение функций между клиентской и пользовательской частями СУБД определяется при установке системы.

Упоминавшиеся выше протоколы удаленного вызова процедур особенно важны в системах управления базами данных, основанных на архитектуре «клиент-сервер».

Во-первых, использование механизма удаленных процедур позволяет действительно перераспределять функции между клиентской и серверной частями системы, поскольку в тексте программы удаленный вызов процедуры ничем не отличается от удаленного вызова и, следовательно, теоретически любой компонент системы может располагаться и на стороне сервера, и на стороне клиента.

Во-вторых, механизм удаленного вызова скрывает различия между взаимодействующими компьютерами. Физически неоднородная локальная сеть компьютеров приводится к логически однородной сети взаимодействующих программных компонентов. В результате пользователи не обязаны серьезно заботиться о разовой закупке совместимых серверов и рабочих станций.

9.4.2 Типичное разделение функций между клиентами и серверами

В типичном на сегодняшний день случае на стороне клиента СУБД работает только такое программное обеспечение, которое не имеет непосредственного доступа к базам данных, а обращается для этого к серверу с использованием языка SQL.

В некоторых случаях хотелось бы включить в состав клиентской части системы некоторые функции для работы с «локальным кэшем» базы данных, т.е. с той ее частью, которая интенсивно используется клиентской прикладной программой. В современной технологии это можно сделать только путем формального создания на стороне клиента локальной копии сервера базы данных и рассмотрения всей системы как набора взаимодействующих серверов.

С другой стороны, иногда хотелось бы перенести большую часть прикладной системы на сторону сервера, если разница в мощности клиентских рабочих станций и сервера велика. В общем-то при использовании RPC это сделать нетрудно. Но требуется, чтобы базовое программное обеспечение сервера действительно позволяло это. В частности, при использовании ОС UNIX проблемы практически не возникают.

Требования к аппаратуре и программному обеспечению клиентских и серверных компьютеров различаются в зависимости от вида использования системы.

Если разделение между клиентом и сервером достаточно жесткое (как в большинстве современных СУБД), то пользователям, работающим на рабочих станциях или персональных компьютерах, абсолютно все равно, какая аппаратура и операционная система работают на сервере, лишь бы он справлялся с возникающим потоком запросов.

Но если могут возникнуть потребности перераспределения функций между клиентом и сервером, то уже совсем не все равно, какие операционные системы используются.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основная литература

1. Карпова, Т. Базы данных: модели, разработка, реализация: учебник / Т. Карпова. – СПб.: Питер, 2001. – 465 с.
2. Глушаков, С.В. Базы данных: учебный курс / С.В. Глушаков, Д.В. Ломотько. – Киев: Фолио, 2000. – 512 с.
3. Хомоненко, А.Д. Базы данных: учебник для высших учебных заведений / А.Д. Хомоненко, В.М. Цыганков, М.Г. Мальцев. – СПб.: КОРОНА, 2002. – 572 с.
4. Дейт, К. Дж. Введение в системы баз данных: пер с англ. – 6-е изд. / К. Дж. Дейт. – СПб.: Вильямс, 2001. – 1072 с.

Дополнительная литература

5. Кренке, Д. Теория и практика построения баз данных / Д. Кренке. – СПб.: Питер, 2003. – 800 с.
6. Когаловский, М.Р. Энциклопедия технологий баз данных / М.Р. Когаловский. – М.: Финансы и статистика, 2002. – 800 с.
7. Ролланд, Ф.Д. Основные концепции баз данных / Ф.Д. Ролланд. – СПб.: Вильямс, 2002. – 256 с.
8. Фролов, А. Базы данных в Интернете: Практическое руководство по созданию Web-приложений с базами данных / А. Фролов, Г. Фролов. – М.: Русская редакция, 2002. – 432 с.
9. Гарсиа-Молина, Г. Системы баз данных: полный курс / Г. Гарсиа-Молина, Дж. Ульман, Дж. Уидом. – СПб.: Вильямс, 2001. – 1086 с.
10. Коннолли, Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика / Т. Коннолли, К. Бегг, А. Страчан. – СПб.: Вильямс, 2000. – 1120 с.
11. Горев, А. Эффективная работа с СУБД / А. Горев, Р. Ахаян, С. Макашарипов. – СПб.: Питер, 1997. – 704 с.
12. Диго, С.М. Проектирование и использование баз данных: учебник / С.М. Диго. – М.: Финансы и статистика, 1995. – 208 с.
13. Грофф, Дж.Р. Энциклопедия SQL: наиболее полное и подробное руководство: пер. с англ. / Дж.Р. Грофф, П.Н. Вайнберг. – 3-е изд. – СПб.: Питер, 2004. – 896 с.
14. Мамаев, Е. SQL Server 2000 в подлиннике / Е. Мамаев. – СПб.: БХВ-Петербург, 2001. – 1280 с.