(1)

Файловые системы.

С точки зрения прикладной программы *файл* – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Термин *файловая система (file system)* используется для обозначения программной системы, управляющей файлами, и архива файлов, хранящегося во внешней памяти. Первая развитая файловая система (OS/360) была разработана специалистами IBM в середине 60-х гг. для выпускавшейся компанией серии компьютеров «360». В этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы, а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами.

Особенности организации устройств внешней памяти на магнитных дисках.

Аппаратура магнитных дисков допускает выполнение обмена с дисками порциями данных произвольного размера. Однако возможность обмениваться с магнитными дисками порциями, размеры которых меньше полного объема блока, в настоящее время в файловых системах не используется. Это связано с двумя обстоятельствами.

Во-первых, считывание или запись только части блока не приводит к существенному выигрышу в суммарном времени обмена. Во-вторых, для работы с частями блоков файловая система должна обеспечить буферы оперативной памяти соответствующего размера, что существенно усложняет распределение оперативной памяти. Алгоритмы распределения памяти порциями произвольного размера плохи тем, что любой из них рано или поздно приводит к *внешней фрагментации* памяти.

Структуры файлов на дисках.

Существуют два основных подхода. При первом подходе, свойственном, например, файловым системам операционных систем компании DEC RSX и VMS, пользователи представляют файл как последовательность записей. Каждая запись – это последовательность байтов, имеющая постоянный или переменный размер. Можно читать или писать записи последовательно либо позиционировать файл на запись с указанным номером. Некоторые файловые системы позволяют структурировать записи на поля и объявлять некие поля ключами записи.

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

Способы организации архивов файлов.

Во всех современных файловых системах обеспечивается многоуровневое именование файлов за счет наличия во внешней памяти каталогов – дополнительных файлов со специальной структурой. Каждый каталог содержит имена каталогов и/или файлов, хранящихся в данном каталоге. Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл.

Поддержка многоуровневой схемы именования файлов обеспечивает несколько преимуществ, основным из которых является простая и удобная схема логической классификации файлов и генерации их имен.

Принципы именования.

Разница между способами именования файлов в разных файловых системах состоит в том, с чего начинается эта цепочка имен. Имеются два радикально различных подхода.

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

Другой крайний вариант был реализован в файловых системах операционной системы Multics. Пользователям обеспечивалась возможность представлять всю совокупность каталогов и файлов в виде единого дерева. Полное имя файла начиналось с имени корневого каталога, и пользователь не обязан был заботиться об установке на дисковое устройство каких-либо конкретных дисков. Сама система, выполняя поиск файла по его имени, запрашивала у оператора установку необходимых дисков. Такую файловую систему можно назвать полностью *централизованной*.

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

(2)

Способы авторизации доступа к файлам.

В большинстве современных систем управления файлами применяется подход к защите файлов, впервые реализованный в ОС UNIX (так называемый *дискреционный* подход). В этой системе каждому зарегистрированному пользователю соответствует пара целочисленных идентификаторов: идентификатор группы, к которой относится пользователь, и его собственный идентификатор. Этими же идентификаторами снабжается каждый процесс, запущенный от имени данного пользователя и имеющий возможность обращаться к системным вызовам файловой системы. Соответственно, при каждом файле хранится полный идентификатор пользователя (собственный идентификатор плюс идентификатор группы), который создал этот файл, и помечается, какие действия с файлом может производить он сам, какие действия с файлом доступны для остальных пользователей той же группы и что могут делать с файлом пользователи других групп. Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Хранимая информация очень компактна (два целых числа для представления идентификаторов и шкала из 9 бит для характеристики возможных действий), при проверке требуется небольшое количество действий, и этот способ контроля доступа в большинстве случаев удовлетворителен.

Организация мультидоступа.

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

(3)

Области применения файловых систем.

Файловые системы обычно обеспечивают хранение слабо структурированной информации, оставляя дальнейшую структуризацию прикладным программам. В примерах использования файлов на рис. 1.3 это даже хорошо, потому что при разработке любой новой прикладной системы, опираясь на простые, стандартные и сравнительно дешевые средства файловой системы, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.

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

Требования к базам данных со стороны информационных систем:

согласованность данных

Система должна учитывать, что в ряде случаев изменение данных в одном файле должно автоматически вызывать модификацию второго файла, чтобы общее содержимое файлов было согласованным. Например, если на работу принимается новый служащий, то нужно добавить запись в файл СЛУЖАЩИЕ, а также должным образом изменить поле РАЗМЕР_ОТДЕЛА в записи файла ОТДЕЛЫ, соответствующей отделу этого служащего.

Понятие согласованности, или целостности, данных является ключевым понятием баз данных. Фактически, если информационная система поддерживает согласованное хранение данных в нескольких файлах, можно говорить о том, что она поддерживает базу данных (БД). Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее *системой управления базами данных* (*СУБД*).

языки запросов

Даже в нашем примере пользователю информационной системы будет не слишком просто получить, например, общую численность отдела, в котором работает Петр Иванович Сидоров.

Придется сначала узнать номер отдела, в котором работает указанный служащий, а затем установить численность этого отдела. Было бы гораздо проще, если бы СУБД позволяла сформулировать такой запрос на языке, более близком пользователям. Такие языки называются *языками запросов к базам данных*.

восстановление согласованного состояния после сбоев

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

Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии. Потребуется выяснить это (а для этого нужно явно проверить соответствие данных в файлах СЛУЖАЩИЕ и ОТДЕЛЫ) и привести данные в согласованное состояние. Настоящие СУБД берут такую работу на себя, поддерживая транзакционное управление и журнализацию изменений базы данных. Прикладная система не обязана заботиться о поддержке корректности состояния базы данных, хотя и должна знать, какие цепочки операций изменения данных являются допустимыми.

реальный режим мультидоступа

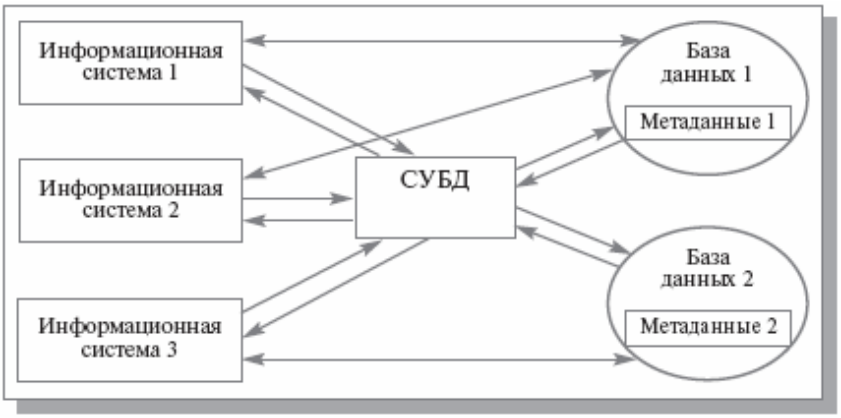

Рис. 1.9. Отдельная СУБД и базы данных с метаданными

Здесь мы видим три информационные системы, которые через одну СУБД работают с двумя разными базами данных, причем первая и вторая системы работают с общей базой данных. Это возможно, поскольку метаданные каждой базы данных содержатся в самих базах данных, и достаточно лишь указать СУБД, с какой базой данных желает работать данное приложение. Поскольку СУБД функционирует отдельно от приложений, и ее работа с базами данных регулируется метаданными, совместное использование одной базы данных двумя информационными системами не вызовет потери согласованности данных, и доступ к данным будет должным образом синхронизироваться. Заметим, что рис. 1.9 вплотную приближает нас к наиболее распространенной в последние десятилетия архитектуре «клиент-сервер». СУБД играет роль «сервера», обсуживающего нескольких «клиентов» – прикладных информационных систем.

Таким образом, СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании файловых систем. При этом существуют приложения, для которых вполне достаточно файлов; приложения, для которых необходимо решать, какой уровень

работы с данными во внешней памяти для них требуется, и приложения, для которых, безусловно, нужны базы данных.

(4)

Основные функции СУБД. Типовая организация СУБД.

В *модели данных* описывается 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.

Согласно Дейту реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода: *структурной части*, *манипуляционной части* и *целостной части*.

В *структурной части* модели данных фиксируются основные логические структуры данных, которые могут применяться на уровне пользователя при организации БД, соответствующих данной модели.

Манипуляционная часть модели данных содержит спецификацию одного или нескольких языков, предназначенных для написания запросов к БД.

Наконец, в *целостной части* модели данных (которая явно выделяется не во всех известных моделях) специфицируются механизмы ограничений целостности, которые обязательно должны поддерживаться во всех реализациях СУБД, соответствующих данной модели.

(5)

Дореляционные модели данных.

В целом ранние системы можно охарактеризовать следующим образом:

- Эти системы активно использовались в течение многих лет, задолго до появления работоспособных 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.
- 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.
- Можно считать, что уровень средств ранних СУБД соотносится с уровнем файловых систем примерно так же, как уровень языка Cobol соотносится с уровнем языков ассемблера. Заметим, что при таком взгляде уровень реляционных систем соответствует уровню языков Ада или APL.
- Навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователей самих производить всю оптимизацию доступа к БД, без какой-либо поддержки системы.
- После появления реляционных систем большинство ранних систем было оснащено

«реляционными» интерфейсами. Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме.

1. Модель данных инвертированных таблиц

Организация доступа к данным на основе инвертированных таблиц используется практически во всех современных реляционных СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным таблицам (индексам).

a) Структуры данных

База данных в модели инвертированных таблиц похожа на БД в модели SQL, но с тем отличием, что пользователям видны и хранимые таблицы, и пути доступа к ним. При этом:

- Строки таблиц упорядочиваются системой в некоторой физической, видимой пользователям 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.

b) Манипулирование данными

Поддерживаются два класса операций:

1. Операции, устанавливающие адрес записи и разбиваемые на два подкласса:
   - прямые поисковые операторы (например, установить адрес первой записи таблицы по некоторому пути доступа);
   - операторы, устанавливающие адрес записи при указании относительной позиции от предыдущей записи по некоторому пути доступа.
2. Операции над адресуемыми записями.

c) Ограничения целостности

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

2. Иерархическая модель данных

a)Структуры данных

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.

Рис. 2.1. Пример типа дерева

Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для иерархической базы данных определяется полный порядок обхода дерева: сверху-вниз, слева-направо.

b) Манипулирование данными

Примеры:

- найти указанный экземпляр типа дерева БД (например, отдел 310);
- перейти от одного экземпляра типа дерева к другому;
- перейти от экземпляра одного типа записи к экземпляру другого типа записи внутри дерева (например, перейти от отдела к первому сотруднику);
- перейти от одной записи к другой в порядке обхода иерархии;
- вставить новую запись в указанную позицию;
- удалить текущую запись.

c) Ограничения целостности

В иерархической модели данных автоматически поддерживается целостность ссылок между предками и потомками. Основное правило: *никакой потомок не может существовать без своего родителя.* Заметим, что аналогичная поддержка целостности по ссылкам между записями без связи «предок-потомок», не обеспечивается. Примером такой «внешней» ссылки является содержимое поля Рук_Отдел в экземпляре типа записи Руководитель.

3. Сетевая модель данных

a)Структура данных

Сетевой подход к организации данных является расширением иерархического подхода. В иерархических структурах запись-потомок должна иметь в точности одного предка; в сетевой структуре данных у потомка может иметься 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 записи *P* является предком только в одном экземпляре типа связи *L*;
- каждый экземпляр типа записи *C* является потомком не более чем в одном экземпляре типа связи *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; и 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.

Рис. 2.3. Пример схемы сетевой базы данных

b) Манипулирование данными

Примерный набор операций манипулирования данными:

- найти конкретную запись в наборе однотипных записей (например, служащего с именем Иванов);
- перейти от предка к первому потомку по некоторой связи (например, к первому служащему отдела 625);
- перейти к следующему потомку в некоторой связи (например, от Иванова к Сидорову);
- перейти от потомка к предку по некоторой связи (например, найти отдел, в котором работает Сидоров);
- создать новую запись;
- уничтожить запись;
- модифицировать запись;
- включить в связь;
- исключить из связи;
- переставить в другую связь и т.д.

c) Ограничения целостности

Имеется (необязательная) возможность потребовать для конкретного типа связи отсутствие потомков, не участвующих ни в одном экземпляре этого типа связи (как в иерархической модели).

(6)

Основные черты модели данных SQL.

Модель данных SQL в относительно законченном виде сложилась к 1999 г., когда был принят и опубликован стандарт SQL:1999.

a) Структура данных

SQL-ориентированная база данных представляет собой набор таблиц, каждая из которых в любой момент времени содержит некоторое мультимножество строк, соответствующих заголовку таблицы. В этом состоит первое и наиболее важное отличие модели данных SQL от реляционной модели данных. Вторым существенным отличием является то, что для таблицы поддерживается порядок столбцов, соответствующий порядку их определения. Другими словами, таблица – это вовсе не отношение, хотя во многом они похожи.

Имеется две основных разновидности таблиц, хранимых в базе данных: *традиционная таблица* и *типизированная таблица*.

Традиционная таблица определяется как множество столбцов с указанными типами данных. В SQL поддерживаются следующие категории типов данных: точные числовые типы; приближенные числовые типы; типы символьных строк; типы битовых строк; типы даты и времени; типы временных интервалов; булевский тип; типы коллекций; анонимные строчные типы; типы, определяемые пользователем; ссылочные типы.

При определении*типизированной таблицы* указывается ранее определенный структурный тип, и если в нем содержится *n* атрибутов, то в таблице образуется *n*+1 столбец, из которых последние *n* столбцов с именами и типами данных, совпадающими именам и типам атрибутов структурного типа. Первый же столбец, имя которого явно задается, называется «самоссылающимся» и содержит типизированные уникальные идентификаторы строк, которые могут генерироваться системой при вставке строк в типизированную таблицу, явно указываться пользователями или состоять из комбинации значений других столбцов. Типом «самоссылающегося» столбца является ссылочный тип, ассоциированный со структурным типом типизированной таблицы. Способ генерации значений ссылочного типа указывается при определении соответствующего структурного типа и подтверждается при определении типизированной таблицы.

b) Манипулирование данными

Средства манипулирования данными составляют значительную часть языка SQL. Здесьжемыограничимсяобщейхарактеристикойоператора SQL SELECT, предназначенногодлявыборкиданныхиимеющегоследующийсинтаксис: *SELECT [ALL | DISTINCT] select_item_commalist FROM table_reference_commalist [WHERE conditional_expression] [GROUP BY column_name_commalist] [HAVING conditional_expression] [ORDER BY order_item_commalist].*

Выборка данных производится из одной или нескольких таблиц, указываемых в разделе FROM запроса. В последнем случае на первом этапе выполнения оператора SELECT образуется одна общая таблица, получаемая из исходных таблиц путем применения операции расширенного декартова умножения. Таблицы могут быть как базовыми, реально хранимыми в базе данных (традиционными или типизированными), так и порожденными, т.е. задаваемыми в виде некоторого оператора SELECT. Это допускается, поскольку результатом выполнения оператора SELECT в его базовой форме является традиционная таблица. Кроме того, в разделе FROM можно указывать выражения соединения базовых и/или порожденных таблиц, результатами которых опять же являются традиционные таблицы.

На следующем шаге общая таблица, полученная после выполнения раздела, подвергается фильтрации путем вычисления для каждой ее строки логического выражения, заданного в разделе WHERE запроса. В отфильтрованной таблице остаются только те строки общей таблицы, для которых значением логического выражения является *true*.

Если в операторе отсутствует раздел GROUP BY, то после этого происходит формирование результирующей таблицы запроса путем вычисления выражений, заданных в списке выборки оператора SELECT. В этом случае список выборки вычисляется для каждой строки отфильтрованной таблицы, и в результирующей таблице появится ровно столько же строк.

При наличии раздела GROUP BY из отфильтрованной таблицы получается сгруппированная таблица, в которой каждая группа состоит из кортежей отфильтрованной таблицы с одинаковыми значениями столбцов группировки, задаваемых в разделе GROUP BY. Если в запросе отсутствует раздел HAVING, то результирующая таблица строится прямо на основе сгруппированной таблицы. Иначе образуется отфильтрованная сгруппированная таблица, содержащая только те группы, для которых значением логического выражения, заданного в разделе HAVING, является *true*.

Результирующая таблица на основе сгруппированной или отфильтрованной сгруппированной таблицы строится путем вычисления списка выборки для каждой группы. Тем самым, в результирующей таблице появится ровно столько строк, сколько групп содержалось в сгруппированной или отфильтрованной сгруппированной таблице.

Если в запросе присутствует ключевое слово DISTINCT, то из результирующей таблицы устраняются строки-дубликаты, т.е. запрос вырабатывает не мультимножество, а множество строк.

Наконец, в запросе может присутствовать еще и раздел ORDER BY. В этом случае результирующая таблица сортируется в порядке возрастания или убывания в соответствии со значениями ее столбцов, указанных в разделе ORDER BY. Результатом такого запроса является не таблица, а отсортированный список, который нельзя сохранить в базе данных. Сам же запрос, содержащий раздел ORDER BY, нельзя использовать в разделе FROM других запросов.

Приведенная характеристика средств манипулирования данными языка SQL является не вполне точной и полной. Кроме того, она отражает *семантику* оператора SQL, а не то, как он обычно исполняется в SQL-ориентированных СУБД.

c) Ограничения целостности

Как отмечалось в начале этого раздела, наиболее важным отличием модели данных SQL от реляционной модели данных является то, что таблицы SQL могут содержать мультимножества строк. Из этого, в частности, следует, что в модели SQL отсутствует обязательное предписание об ограничении целостности сущности. В базе данных могут существовать таблицы, для которых не определен первичный ключ. С другой стороны, если для таблицы определен первичный ключ, то для нее ограничение целостности сущности поддерживается точно так же, как это требуется в реляционной модели данных.

Ссылочная целостность в модели данных SQL поддерживается в обязательном порядке, но в трех разных вариантах, лишь один из которых полностью соответствует реляционной модели. Это связано с уже упоминавшимся в этом разделе интенсивным использованием в SQL неопределенных значений. Подробнее особенности ограничений ссылочной целостности в SQL рассматриваются в лекции 16.

Кроме того, в SQL имеются развитые возможности явного определения ограничений целостности на уровне столбцов таблиц, на уровне таблиц целиком и на уровне базы данных.

(7)

Типы данных в SQL.

В SQL поддерживаются следующие категории типов данных: точные числовые типы; приближенные числовые типы; типы символьных строк; типы битовых строк; типы даты и времени; типы временных интервалов; булевский тип; типы коллекций; анонимные строчные типы; типы,

определяемые пользователем; ссылочные типы.

Булевский тип в SQL содержит три значения – *true*, *false* и *uknown*. Это связано с интенсивным использованием в SQL так называемого неопределенного значения (NULL), которое разрешается использовать вместо значения любого типа данных.

В модели данных SQL допускается объявление двух видов *типов коллекций*: *типы массива* и *типы мультимножества*. Элементы типа коллекции могут быть любого типа данных, определенного к моменту определения данного типа коллекции. При объявлении типа мультимножества можно явно запретить наличие в его значениях элементов-дубликатов, что фактически приводит к объявлению типа множества.

Анонимный строчный тип – это безымянный структурный тип, значения которого являются строками, состоящими из элементов ранее определенных типов.

Поддерживается два вида типов данных, определяемых пользователями: *индивидуальные* и *структурные типы*. Индивидуальный тип – это именованный тип данных, основанный на единственном предопределенном типе. Индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Чтобы выполнить некоторую операцию базового типа над значениями определенного над ним индивидуального типа, требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями базового типа. Имеется также возможность явного определения методов, функций и процедур, связанных с данным индивидуальным типом.

Структурный тип данных – это именованный тип данных, включающий один или более атрибутов любого из допустимых в SQL типа данных, в том числе другого структурного типа, типа коллекций, анонимного строчного типа и т. д. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа. При определении структурного типа можно использовать механизм *наследования* от ранее определенного структурного типа.

Наследование типов в SQL.

При определении типизированных таблиц можно использовать *механизм наследования*. Можно определить подтаблицу типизированной супертаблицы, если структурный тип подтаблицы является непосредственным подтипом структурного типа супертаблицы. Подтаблица наследует у супертаблицы способ генерации значений ссылочного типа и все ограничения целостности, которые были специфицированы в определении супертаблицы. Дополнительно можно определить ограничения, затрагивающие новые столбцы.

(8)

Основные черты модели данных ODMG.

a) Структура данных

Объектно-ориентированная модель данных, специфицированная в ODMG 3.0, отличается от других двух моделей, описываемых в этом разделе, прежде всего, в одном принципиальном аспекте. В модели данных SQL и истинной реляционной модели данных база данных представляет собой набор именованных контейнеров данных одного родового типа: таблиц или отношений соответственно. В объектно-ориентированной модели данных база данных – это набор объектов (контейнеров данных) произвольного типа.

b) Манипулирование данными

В стандарте ODMG в качестве базового средства манипулирования объектными базами данных

предлагается язык OQL (Object Query Language). Это небольшой, но достаточно сложный язык запросов. Разработчики в целом характеризуют его следующим образом:

- OQL опирается на объектную модель ODMG (имеется в виду, что в нем поддерживаются средства доступа ко всем возможным структурам данных, допускаемых в структурной части модели).
- OQL очень близок к SQL/92. Расширения относятся к объектно-ориентированным понятиям, таким как сложные объекты, объектные идентификаторы, путевые выражения, полиморфизм, вызов операций и отложенное связывание.
- В OQL обеспечиваются высокоуровневые примитивы для работы с множествами объектов, но, кроме того, имеются настолько же эффективные примитивы для работы со структурами, списками и массивами.
- OQL является функциональным языком, допускающим неограниченную композицию операций, если операнды не выходят за пределы системы типов. Это является следствием того факта, что результат любого запроса обладает типом, принадлежащим к модели типов ODMG, и поэтому к результату запроса может быть применен новый запрос.
- OQL не является вычислительно полным языком. Он представляет собой простой язык запросов.
- Операторы языка OQL могут вызываться из любого языка программирования, для которого в стандарте ODMG определены правила связывания. И, наоборот, в запросах OQL могут присутствовать вызовы операций, запрограммированных на этих языках.
- В OQL не определяются явные операции обновления, а используются вызовы операций, определенных в объектах для целей обновления.
- В OQL обеспечивается декларативный доступ к объектам. По этой причине OQL-запросы могут хорошо оптимизироваться.
- Можно легко определить формальную семантику OQL.

В совокупности результатом допустимых в OQL выражений запросов могут являться:

- коллекция объектов;
- индивидуальный объект;
- коллекция литеральных значений;
- индивидуальное литеральное значение.

c) Ограничения целостности

В соответствии с общей идеологией объектно-ориентированного подхода в модели ODMG два объекта считаются совпадающими в том и только в том случае, когда являются одним и тем же объектом, т.е. имеют один и тот же OID. Объекты одного объектного типа с разными OID считаются разными, даже если обладают полностью совпадающими состояниями. Поэтому в объектной модели отсутствует аналог ограничения целостности сущности реляционной модели данных.

Что же касается ссылочной целостности, то она поддерживается, если между двумя атомарными объектными типами определяется связь вида «один-ко-многим». В этом случае объекты на стороне связи «один» рассматриваются как предки, а объекты на стороне связи «многие» – как потомки, и ООСУБД обязана следить за тем, чтобы не образовывались потомки без предков.

(9)

Типы данных, наследование типов в модели данных ODMG.

В объектной модели данных вводятся две разновидности типов: *литеральные* и *объектные* типы.

Литеральные типы данных – это обычные типы данных, принятые в традиционных языках программирования. Они подразделяются на базовые скалярные числовые типы, символьные и булевские типы (*атомарные литералы*), конструируемые типы записей (*структур*) и *коллекций*.

Объектные типы в объектной модели данных по смыслу ближе всего к понятию *класса* в объектно-ориентированных языках программирования. У каждого объектного типа имеется операция создания и инициализации нового объекта этого типа. Эта операция возвращает значение ObjectID (OID) нового объекта, который можно хранить в любом месте, где допускается хранение объектов данного типа, и использовать для обращения к *операциям* объекта, определенным в его объектном типе. Имеются два вида объектных типов.

Первый из них называется атомарным объектным типом. Нестрого говоря, при определении атомарного объектного типа указывается его внутренняя структура (набор свойств – атрибутов и связей) и набор операций, которые можно применять к объектам этого типа. Для определения атомарного объектного типа можно использовать механизм наследования, расширяя набор свойств и/или переопределяя существующие и добавляя новые операции.

Второй вид – это объектные типы коллекций. Как и в случае использования литеральных типов коллекций, можно определять объектные типы множеств, мультимножеств, списков и словарей. Типом элемента объектного типа коллекции может быть любой литеральный или объектный тип за исключением того же типа коллекции. У объектных типов коллекций имеются предопределенные наборы операций. В отличие от литеральных типов коллекций, которые, как и все литеральные типы являются множествами значений, объектные типы коллекций обладают операцией создания объекта, имеющего, как и все объекты, собственный OID.

(10)

Основные черты истинно реляционной модели данных.

a) Структура данных

База данных в истинной реляционной модели – это набор долговременно хранимых именованных *переменных отношений*, каждая из которых определена на некотором типе отношений. В каждый момент времени каждая переменная отношения базы данных содержит некоторое значение отношения соответствующего типа.

b) Манипулирование данными

Вообще говоря, в качестве эталонного средства манипулирования данными в истинной реляционной модели можно использовать алгебру Кодда. Однако Дейт и Дарвен предложили новую реляционную алгебру, названную ими *Алгеброй A*, которая основывается на реляционных аналогах булевских операций *конъюнкции*, *дизъюнкции* и *отрицания*. Через ее операции выражаются все операции алгебры Кодда.

c) Ограничения целостности

В число обязательных требований истинной реляционной модели входит требование определения хотя бы одного возможного ключа для каждой переменной отношения (возможный ключ – это одно из подмножеств заголовка переменной отношения, обладающее свойствами первичного ключа). Кроме того, говорится, что «любое условное выражение, которое является (или логически эквивалентно) замкнутой правильно построенной формулой (WFF) реляционного исчисления, должно быть допустимо в качестве спецификации ограничения целостности».

Средства поддержки декларативной ссылочной целостности фигурируют только в разделе

рекомендуемых возможностей: «В конкретную реализацию истинной реляционной модели следует включить некоторую декларативную сокращенную форму для выражения ссылочных ограничений (называемых также ограничениями внешнего ключа)».

(11)

Типы данных в истинно реляционной модели данных.

В истинно реляционной модели очень большое внимание уделяется типам данных. Предлагаются три категории типов данных: *скалярные типы*, *кортежные типы* и *типы отношений*.

Скалярный тип данных – это привычный инкапсулированный тип, реальная внутренняя структура которого скрыта от пользователей. Предлагаются механизмы определения новых скалярных типов и операций над ними. Типом атрибута определяемого скалярного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Некоторые базовые скалярные типы данных должны быть предопределены в системе. В число этих типов должен входить тип `truth value` (так Дейт и Дарвен называют булевский тип) ровно с двумя значениями *`true`* и *`false`*.

Кортежный тип – это безымянный тип данных, определяемый с помощью генератора типа `TUPLE` с указанием множества пар `<имя_атрибута, тип_атрибута>` (заголовка кортежа). Типом атрибута кортежного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Значением кортежного типа является кортеж, представляющий собой множество триплетов `<имя_атрибута, тип_атрибута, значение_атрибута>`, которое соответствует заголовку кортежа этого кортежного типа.

Тип отношения – это безымянный тип данных, определяемый с помощью генератора типа `RELATION` с указанием некоторого заголовка кортежа. Значением типа отношения является заголовок отношения, совпадающий с заголовком кортежа этого типа отношения, и тело отношения, представляющее собой множество кортежей, соответствующих этому заголовку. Кортежные типы и типы отношений не являются инкапсулированными: имеется возможность прямого доступа к атрибутам.

Наследование типов данных в истинно реляционной модели данных.

Для всех разновидностей типов данных разработана модель множественного наследования, позволяющая определять новые типы данных на основе уже определенных типов. Модель наследования по Дейту и Дарвену не является частью истинной реляционной модели данных.

Понятно, что при таких определениях значениями атрибутов отношения могут быть не только значения произвольно сложных скалярных типов, типами атрибутов которых могут быть, в частности, отношения, но и просто отношения.

(12)

Общие понятия реляционного подхода к организации БД.

Основные понятия реляционных баз данных: тип данных, домен, атрибут, кортеж, отношение, первичный ключ.

Для начала покажем смысл этих понятий на примере отношения СЛУЖАЩИЕ, содержащего информацию о служащих некоторого предприятия (рис. 3.1).

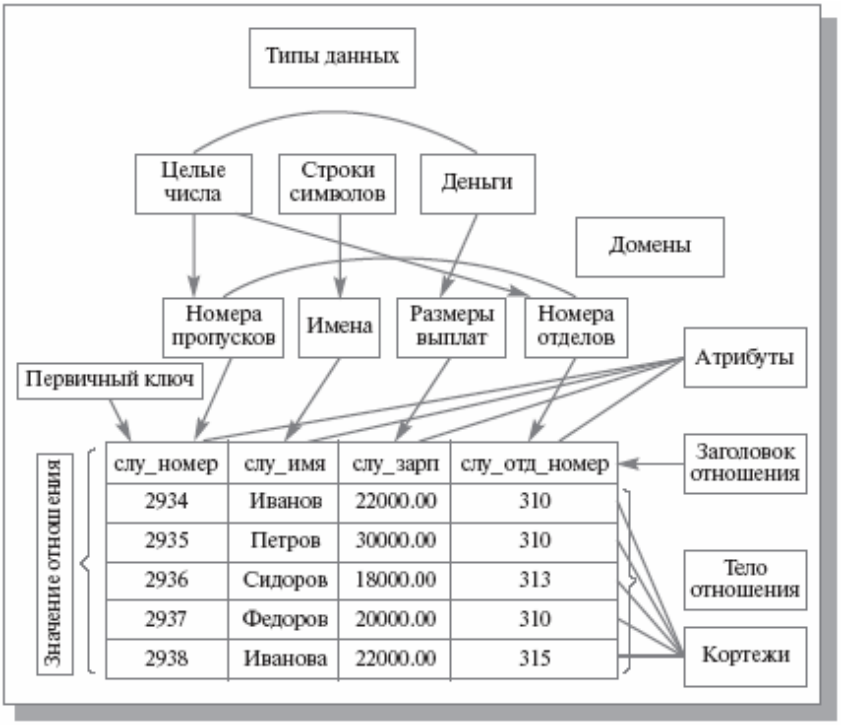

Рис. 3.1. Соотношение основных понятий реляционного подхода

Основные концепции и термины.

1. Тип данных

Значения данных, хранимые в реляционной базе данных, являются типизированными, т. е. известен тип каждого хранимого значения. Понятие типа данных в реляционной модели данных полностью соответствует понятию типа данных в языках программирования.

2. Домен

В общем виде домен определяется путем задания некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу этого типа данных (*ограничения домена*). Элемент данных является элементом домена в том и только в том случае, если вычисление этого логического выражения дает результат *истина*. С каждым доменом связывается имя, уникальное среди имен всех доменов соответствующей базы данных.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену.

3. Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения

Заголовком (или *схемой*) *отношения* `r` (`Hr`) называется конечное множество упорядоченных пар вида `<A, T>`, где `A` называется именем атрибута, а `T` обозначает имя некоторого базового типа или ранее определенного домена. По определению требуется, чтобы все имена атрибутов в заголовке отношения были различны.

Кортежем `tr`, соответствующим заголовку `Hr`, называется множество упорядоченных триплетов вида `<A, T, v>`, по одному такому триплету для каждого атрибута в `Hr`. Третий элемент – `v` – триплета `<A, T, v>` должен являться допустимым значением типа данных или домена `T`.

Телом `Br` отношения `r` называется произвольное множество кортежей `tr`.

Значением Vr отношения r называется пара множеств Hr и Br.

По определению, *степенью*, или *«арностью»*, заголовка отношения, кортежа, соответствующего этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения.

При приведенных определениях разумно считать *схемой реляционной базы данных* набор пар <имя_VARr, Hr>, включающий имена и заголовки всех переменных отношения, которые определены в базе данных. *Реляционная база данных* – это набор пар <VARr, Hr> (конечно, каждая переменная отношения в любой момент времени содержит некоторое значение-отношение, в частности, пустое).

4. Первичный ключ и интуитивная интерпретация реляционных понятий

По определению, *первичным ключом* переменной отношения является такое подмножество S множества атрибутов ее заголовка, что в любое время значение первичного ключа (составное, если в состав первичного ключа входит более одного атрибута) в любом кортеже тела отношения отличается от значения первичного ключа в любом другом кортеже тела этого отношения, а никакое собственное подмножество S этим свойством не обладает.

(13)

Фундаментальные свойства отношений.

1. Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений

То свойство, что тело любого отношения никогда не содержит кортежей-дубликатов, следует из определения тела отношения как множества кортежей. В классической теории множеств по определению любое множество состоит из различных элементов.

Именно из этого свойства вытекает наличие у каждого значения отношения первичного ключа – минимального множества атрибутов, являющегося подмножеством заголовка данного отношения, составное значение которых уникально определяет кортеж отношения.Однако в 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 – однозначного определения кортежа.

Забегая вперед, заметим, что во многих практических реализациях реляционных СУБД 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 часто приводит к серьезным проблемам. Мы остановимся на этом подробнее при обсуждении языка SQL.

2. Отсутствие упорядоченности кортежей

Конечно, формально свойство отсутствия упорядоченности кортежей в значении отношения также является следствием определения тела отношения как множества кортежей.

Хранить упорядоченные списки кортежей в условиях интенсивно обновляемой базы данных гораздо сложнее технически, а поддержка упорядоченности влечет за собой существенные накладные расходы.

3. Отсутствие упорядоченности атрибутов

Атрибуты отношений не упорядочены, поскольку по определению заголовок отношения есть множество пар <имя атрибута, имя домена>.

4. Атомарность значений атрибутов, первая нормальная форма отношения

Значения всех атрибутов являются атомарными (вернее, скалярными). Это следует из определения домена как потенциального множества значений скалярного типа данных, т. е. среди значений домена не могут содержаться значения с видимой структурой, в том числе множества значений (отношения).

(14)

Реляционная модель данных: общее понятие и составные части.

Согласно трактовке Дейта, 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 родовой структурой данных, используемой в реляционных БД, является нормализованное n-арное отношение. Определяются понятия доменов, атрибутов, кортежей, заголовка, тела и переменной отношения. По сути дела, в двух предыдущих разделах этой лекции мы рассматривали именно понятия и свойства структурной составляющей реляционной модели.

В *манипуляционной части* модели определяются два фундаментальных механизма манипулирования реляционными БД – *реляционная алгебра* и *реляционное исчисление*. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями и добавлениями), 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 сущности (entity integrity)*.

На самом деле, требование целостности сущности полностью звучит следующим образом: у любой переменной отношения должен существовать первичный ключ, и никакое значение первичного ключа в кортежах значения-отношения переменной отношения не должно содержать неопределенных значений.

Требование целостности по ссылкам, или *требование целостности внешнего ключа*, состоит в том, что для каждого значения внешнего ключа, появляющегося в кортеже значения-отношения ссылающейся переменной отношения, либо в значении-отношении переменной отношения, на которую указывает ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть полностью неопределенным (т. е. ни на что не указывать).

Заметим, что, как и первичный ключ, внешний ключ должен специфицироваться при определении переменной отношения и представляет собой ограничение на допустимые значения-отношения этой переменной. Другими словами, определение внешнего ключа представляет собой определение ограничения целостности базы данных.

(15)

Реляционная алгебра Кодда.

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

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:

- объединения отношений;
- пересечения отношений;
- взятия разности отношений;
- взятия декартова произведения отношений.

Специальные реляционные операции включают:

- ограничение отношения;
- проекцию отношения;
- соединение отношений;
- 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.

Поскольку результатом любой реляционной операции (кроме операции присваивания, которая не вырабатывает значения) является некое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение. В построении реляционного выражения могут участвовать все реляционные операции, кроме операции присваивания. Вычислительная интерпретация реляционного выражения диктуется установленными приоритетами операций:

RENAME≥WHERE = PROJECT≥TIMES = JOIN = INTERSECT = DIVIDEBY≥UNION = MINUS

Операция	Приоритет
RENAME	4
WHERE	3
PROJECT	3
TIMES	2
JOIN	2
INTERSECT	2
DIVIDE BY	2
UNION	1
MINUS	1

Рис. 4.1. Таблица приоритетов операций традиционной реляционной алгебры

(16)

Алгебра А.

В исходный базовый набор операций входят операции реляционного дополнения <NOT>, удаления атрибута <REMOVE>, переименования атрибута <RENAME>, реляционной конъюнкции <AND> и

реляционной дизъюнкции <OR>.

(17)

Полнота алгебры А.

Покажем, что Алгебра А является полной, т. е. на основе введенных операций выражаются все операции алгебры Кодда, рассмотренной в предыдущей лекции.

К настоящему моменту в состав базовых операций Алгебры А входят операция <REMOVE> в качестве аналога операции PROJECT, а также операция переименования атрибутов <RENAME>. UNION является частным случаем операции <OR>, TIMES, INTERSECT и NATURAL JOIN – частные случаи операции <AND>. Нам осталось показать, что через операции Алгебры А выражаются операции взятия разности MINUS, ограничения (WHERE), соединения общего вида (JOIN) и реляционного деления (DIVIDE BY).

r_1 MINUS r_2 = r_1 <AND> <NOT> r_2

Предположим, что мы хотим найти всех служащих с заработной платой, равной 20000.00 руб. Возьмем отношение ЗАРП_20000 {СЛУ_ЗАРП}. Мы видим, что результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_20000 в точности совпадает с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП = 20000.00. Если требуется найти служащих, чья заработная плата превышает 20000.00 руб., то возьмем отношение ЗАРП_БОЛЬШЕ_20000. Тогда снова результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_БОЛЬШЕ_20000.00 будет совпадать с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП > 20000.00.Понятно, что аналогичным образом выражаются через <AND> операции ограничения с условиями вида a comp_op const, в которых comp_op является «<», «≥» или «≤». Некоторый особый случай представляет условие вида a ≠const, и мы проиллюстрируем этот случай на примере запроса «Выбрать всех служащих, не получающих заработную плату в размере 22 000.00 руб.». Возьмем отношение ЗАРП_НЕ_22000. Результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_НЕ_22000 будет совпадать с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП 22000.00.

r_1 DIVIDE BY r_2= (r_1 <REMOVE> B) <AND> <NOT> ((r_2 <AND> (r_1 <REMOVE> B)) <AND> <NOT> r_1) <REMOVE> B)

При наличии того факта, что операция взятия расширенного декартова произведения TIMES является частным случаем операции <AND>, после того как мы научились с помощью Алгебры А выполнять ограничения, становится очевидно, что через операции Алгебры А выражаются и соединения общего вида. В общем случае, чтобы получить результат соединения общего вида произвольных отношений А и В, нужно:

- выполнить над одним из отношений одну или несколько операций <RENAME>, чтобы избавиться от общих имен атрибутов;
- выполнить над полученными отношениями операцию <AND>, производящую расширенное декартово произведение;
- и для полученного отношения выполнить одну или несколько операций <AND> с отношениями-константами, чтобы должным образом ограничить его.

(18)

Избыточность алгебры А.

В формальной математической логике стандартным базисом для выражения всех возможных булевских функций является набор {NOT, AND, OR} (отрицание, дизъюнкция и конъюнкция). Известно, что этот набор традиционен, но избыточен, поскольку верны тождества A AND B ≡NOT

(NOT A OR NOT B) и A OR B≡NOT (NOT A AND NOT B). (Эти тождества легко проверяются по таблицам истинности операций.) Оказывается (и это тоже легко проверить, опираясь на определения операций), что аналогичные тождества справедливы для операций <NOT>, <AND> и <OR> Алгебры А. Тем самым, в наборе базовых операций Алгебры А можно оставить операции <AND> и <NOT> (или <OR> и <NOT>).

Наконец, покажем, что избыточна и операция <RENAME>. Для иллюстрации снова воспользуемся отношением СЛУЖАЩИЕ. Пусть нам нужен результат операции СЛУЖАЩИЕ <RENAME> (ПРО_НОМ, НОМЕР_ПРОЕКТА) (мы по-прежнему предполагаем, что множество значений домена атрибута ПРО_НОМ ограничено значениями, представленными в теле отношения СЛУЖАЩИЕ). Возьмем бинарное отношение ПРО_НОМ_НОМЕР_ПРОЕКТА, где каждый из кортежей содержит два одинаковых значения номера проекта и в тело отношения входят все значения домена атрибута ПРО_НОМ. Тогда вычисление выражения (СЛУЖАЩИЕ <AND> ПРО_НОМ_НОМЕР_ПРОЕКТА) <REMOVE> (ПРО_НОМ) приводит к желаемому результату.

(19)

Реляционное исчисление кортежей.

Чтобы определить переменную СЛУЖАЩИЙ, областью определения которой является отношение СЛУЖАЩИЕ, нужно употребить конструкциюRANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ. Из этого определения следует, что в любой момент времени переменная СЛУЖАЩИЙ представляет некоторый кортеж отношения СЛУЖАЩИЕ. Чтобы сослаться на значение атрибута СЛУ_ИМЯ переменной СЛУЖАЩИЙ, нужно употребить конструкцию СЛУЖАЩИЙ.СЛУ_ИМЯ.

Правильно построенные формулы

Правильно построенная формула (Well-Formed Formula, WFF) служит для выражения условий, накладываемых на кортежные переменные.

Основой WFF являются простые условия, представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или литерально заданных констант). Например, конструкции СЛУЖАЩИЙ.СЛУ_НОМ = 2934 и СЛУЖАЩИЙ.СЛУ_НОМ = ПРОЕКТ.ПРОЕКТ_РУК являются простыми условиями.По определению, простое сравнение является WFF, а WFF, заключенная в круглые скобки, представляет собой простое сравнение. Более сложные варианты WFF строятся с помощью логических связок NOT, AND, OR и IF ... THEN с учетом обычных приоритетов операций (NOT > AND > OR) и возможности расстановки скобок. Допускается использование кванторов существования (EXISTS) и всеобщности (FORALL).

Переменные, входящие в WFF, могут быть свободными или связанными. По определению, все переменные, входящие в WFF, при построении которой не использовались кванторы, являются *свободными*. Фактически, это означает, что если для какого-то набора значений свободных кортежных переменных при вычислении WFF получено значение true, то эти значения кортежных переменных могут входить в результирующее отношение. Если же имя переменной использовано сразу после квантора при построении WFF вида EXISTS var (form) или FORALL var (form), то в этой WFF и во всех WFF, построенных с ее участием, var является *связанной переменной*. Это означает, что такая переменная не видна за пределами минимальной WFF, связавшей эту переменную. При вычислении значения такой WFF используется не одно значение связанной переменной, а вся область ее определения.

Целевые списки и выражения реляционного исчисления

Целевой список определяет набор и имена атрибутов результирующего отношения. Строится из целевых элементов, каждый из которых может иметь следующий вид:

- var.attr, где var – имя свободной переменной соответствующей WFF, а attr – имя атрибута отношения, на котором определена переменная var;
- var, что эквивалентно наличию подсписка var.attr1, var.attr2, ..., var.attrn, где {attr1, attr2, ..., attrn} включает имена всех атрибутов определяющего отношения;
- new_name = var.attr; new_name – новое имя соответствующего атрибута результирующего отношения.

Последний вариант требуется в тех случаях, когда в WFF используется несколько свободных переменных с одинаковой областью определения. Фактически применение целевого списка к области истинности WFF эквивалентно действию алгебраической операции проекции, а последний из приведенных вариантов представляет собой некоторую разновидность алгебраической операции переименования атрибута.

Выражением реляционного исчисления кортежей называется конструкция вида `target_list WHERE WFF`. Значением выражения является отношение, тело которого определяется WFF, а множество атрибутов и их имена – целевым списком.

Пример записи
```
СЛУЖАЩИЕ DIVIDE BY НОМЕРА_ПРОЕКТОВ:
        СЛУ1, СЛУ2 RANGE IS СЛУЖАЩИЕ
        НОМЕР_ПРОЕКТА range is НОМЕРА_ПРОЕКТОВ

        СЛУ1.СЛУ_НОМЕР, СЛУ1.СЛУ_ИМЯ, СЛУ1.СЛУ_ЗАРП
        WHERE FORALL НОМЕР_ПРОЕКТА EXISTS СЛУ2
                        (СЛУ1.СЛУ_НОМЕР = СЛУ2.СЛУ_НОМЕР AND
                        СЛУ1.ПРО_НОМ = НОМЕРА_ПРОЕКТОВ.ПРО_НОМ)
```

(20)

Реляционное исчисление доменов.

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного множества предикатов, позволяющих выражать так называемые *условия членства*. Если R – это n-арное отношение с атрибутами a_1, a_2, ..., a_n, то условие членства имеет вид R (a_{i1} : v_{i1}, a_{i2} : v_{i2}, ..., a_{im} : v_{im}) ($m \leq n$), где v_{ij} – это либо литерально задаваемая константа, либо имя доменной переменной. Условие членства принимает значение `true` в том и только в том случае, если в отношении R существует кортеж, содержащий указанные значения указанных атрибутов. Если v_{ij} – константа, то на атрибут a_{ij} накладывается жесткое условие, не зависящее от текущих значений доменных переменных; если же v_{ij} – имя доменной переменной, то условие членства может принимать разные значения при разных значениях этой переменной.

Примеры:

СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов', СЛУ_ЗАРП:22400.00, ПРО_НОМ:1) примет значение `true` в том и только в том случае, когда в теле отношения СЛУЖАЩИЕ содержится кортеж <2934, 'Иванов', 22400.00, 1>.

СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов', СЛУ_ЗАРП:22400.00, ПРО_НОМ:ПРО_НОМ) будет принимать значение `true` для всех комбинаций явно заданных значений и допустимых значений переменной ПРО_НОМ, которые соответствуют кортежам, входящим в тело отношения СЛУЖАЩИЕ.

Во всех остальных отношениях формулы и выражения исчисления доменов выглядят похожими на формулы и выражения исчисления кортежей. В частности, формулы могут включать кванторы, и различаются свободные и связанные вхождения доменных переменных.

Пример: выдать номера и имена служащих, не получающих минимальную заработную плату:

СЛУ_НОМ, СЛУ_ИМЯ WHERE EXISTS СЛУ_ЗАРП1(СЛУЖАЩИЕ (СЛУ_ЗАРП1) ANDСЛУЖАЩИЕ(СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП) ANDСЛУ_ЗАРП > СЛУ_ЗАРП1).

(21)

Функциональные зависимости.

В значении переменной отношения R атрибут Y функционально зависит от атрибута X в том и только в том случае, если каждому значению X соответствует в точности одно значение Y, обозначается как FD X⟶Y.

Если атрибут A отношения *R* является возможным ключом, то для любого атрибута B этого отношения всегда выполняется FD A⟶B.

FD A⟶B называется *тривиальной*, если A⊇B. Очевидно, что любая тривиальная FD всегда выполняется.

FD A⟶C называется *транзитивной*, если существует такой атрибут B, что имеются функциональные зависимости A⟶B и B⟶C и отсутствует функциональная зависимость C⟶A.

Замыкание множества функциональных зависимостей.

Замыканием множества FD S является множество FD S^+, включающее все FD, логически выводимые из FD множества S.

Аксиомы Армстронга.

Для краткости будем обозначать A UNION B через AB. Тогда:

- если B⊆A, то A⟶B (рефлексивность);
- если A⟶B, то AC⟶BC (пополнение);
- если A⟶B и B⟶C, то A⟶C (транзитивность).

Истинность первой аксиомы Армстронга следует из того, что при B⊆A FD A⟶B является тривиальной.

Справедливость второй аксиомы докажем от противного. Предположим, что FD AC⟶BC не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа t_1 и t_2, такие, что t_1{AC} = t_2{AC} (a), но t_1{BC} ≠ t_2{BC} (b) (здесь t{A} обозначает проекцию кортежа t на множество атрибутов A). По аксиоме рефлексивности из равенства (a) следует, что t_1{A} = t_2{A}. Поскольку имеется FD A⟶B, должно соблюдаться равенство t_1{B} = t_2{B}. Тогда из неравенства (b) следует, что t_1{C} ≠ t_2{C}, что противоречит наличию тривиальной FD AC⟶C. Следовательно, предположение об отсутствии FD AC⟶BC не является верным, и справедливость второй аксиомы доказана.

Аналогично докажем истинность третьей аксиомы Армстронга. Предположим, что FD A⟶C не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа t_1 и t_2, такие, что t_1{A} = t_2{A}, но t_1{C} ≠ t_2{C}. Но из наличия FD A⟶B следует, что t_1{B} = t_2{B}, а потому из наличия FD B⟶C следует, что t_1{C} = t_2{C}. Следовательно, предположение об отсутствии FD A⟶C не является верным, и справедливость третьей аксиомы доказана.

Замыкание множества атрибутов.

Пусть заданы отношение R, множество Z атрибутов этого отношения (подмножество заголовка R, или составной атрибут R) и некоторое множество FD S, выполняемых для R. Тогда *замыканием Z над S* называется наибольшее множество Z^+ таких атрибутов Y отношения R, что FD $Z \rightarrow Y$ входит в S^+.

Суперключом отношения R называется любое подмножество K заголовка R, включающее, по меньшей мере, хотя бы один возможный ключ R.

Одно из следствий этого определения состоит в том, что подмножество K заголовка отношения R является суперключом тогда и только тогда, когда для любого атрибута A (возможно, составного) заголовка отношения R выполняется FD $K \rightarrow A$. В терминах замыкания множества атрибутов K является суперключом тогда и только тогда, когда K^+ совпадает с заголовком R.

Минимальное покрытие множества функциональных зависимостей.

Множество FD $S2$ называется *покрытием множества FD*$S1$, если любая FD, выводимая из $S1$, выводится также из $S2$.

Легко заметить, что $S2$ является покрытием $S1$ тогда и только тогда, когда $S1^+ \subseteq S2^+$. Два множества FD $S1$ и $S2$ называются *эквивалентными*, если каждое из них является покрытием другого, т. е. $S1^+ = S2^+$.

Множество FD S *называется минимальным* в том и только в том случае, когда удовлетворяет следующим свойствам:

- правая часть любой FD из S является множеством из одного атрибута (простым атрибутом);
- детерминант каждой FD из S обладает свойством *минимальности*; это означает, что удаление любого атрибута из детерминанта приводит к изменению замыкания S^+, т. е. порождению множества FD, не эквивалентного S;
- удаление любой FD из S приводит к изменению S^+, т. е. порождению множества FD, не эквивалентного S.

Минимальным покрытием множества FD S называется любое минимальное множество FD $S1$, эквивалентное S.

Поскольку для каждого множества FD существует эквивалентное минимальное множество FD, у каждого множества FD имеется хотя бы одно минимальное покрытие, причем для его нахождения не обязательно находить замыкание исходного множества.

(22)

Декомпозиция без потерь и функциональные зависимости.

Декомпозиция – разбиение путем проецированияотношения, находящегося в предыдущей нормальной форме, на два или более отношений, удовлетворяющих требованиям следующей нормальной формы.

Считаются правильными такие декомпозиции отношения, которые обратимы, т. е. имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации. Такие декомпозиции называются *декомпозициями без потерь*.

Теорема Хита.

Пусть задано отношение r {A, B, C} (A, B и C, в общем случае, являются составными атрибутами)

и выполняется FD A⟶B. Тогда*r* = (*r*PROJECT {A, B}) NATURALJOIN (*r*PROJECT {A, C}).

Доказательство. Прежде всего, докажем, что в теле результата естественного соединения (обозначим этот результат через r_1) содержатся все кортежи тела отношения *r*. Действительно, пусть кортеж {a, b, c} $\in r$. Тогда по определению операции взятия проекции {a, b} $\in$(*r* PROJECT {A, B}) и {a, c} $\in$(r PROJECT {A, C}). Следовательно, {a, b, c} $\in r_1$. Теперь докажем, что в теле результата естественного соединения нет лишних кортежей, т. е. что если кортеж {a, b, c} $\in r1$, то {a, b, c} $\in r$. Если {a, b, c} $\in r_1$, то существуют {a, b} $\in$(*r* PROJECT {A, B}) и {a, c} $\in$(*r* PROJECT {A, C}). Последнее условие может выполняться в том и только в том случае, когда существует кортеж {a, b*, c} $\in r$. Но поскольку выполняется FD A⟶B, то b = b* и, следовательно, {a, b, c} = {a, b*, c}.

(23)

Проектирование реляционных баз данных с использованием нормализации.

В основе процесса проектирования лежит метод *нормализации*, т. е. декомпозиции отношения, находящегося в предыдущей *нормальной форме*, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы.

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений.

Основные свойства нормальных форм состоят в следующем:

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

Первая нормальная форма.

Значения всех атрибутов отношения атомарны.

Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, будем считать, что исходный набор отношений уже соответствует этому требованию.

Вторая нормальная форма.

1NF + каждый неключевой атрибут (не входящий ни в один возможный ключ) минимально функционально зависит от первичного ключа(предполагается, что у отношения имеется только один возможный ключ).

Третья нормальная форма.

2NF + каждый неключевой атрибут нетранзитивно функционально зависит от первичного ключа.

(24)

Теорема Риссонена.

Проекции r_1 и r_2 отношения r являются *независимыми* тогда и только тогда, когда:

- каждая FD в отношении r логически следуетиз FD в r_1 и r_2;
- общие атрибуты r_1 и r_2 образуют возможный ключ хотя бы для одного из этих отношений.

Атомарным отношением называется отношение, которое невозможно декомпозировать на независимые проекции.

Нормальная форма Бойса-Кодда.

Переменная отношения находится в *нормальной форме Бойса-Кодда (BCNF)* в том и только в том случае, когда любая выполняемая для этой переменной отношения нетривиальная и минимальная FD имеет в качестве детерминанта некоторый возможный ключ данного отношения.

(25)

Многозначные зависимости.

В переменной отношения R с атрибутами A, B, C (в общем случае, составными) имеется *многозначная зависимость* (multi-valued dependency – MVD) B от A ($A \twoheadrightarrow B$) в том и только в том случае, когда множество значений атрибута B, соответствующее паре значений атрибутов A и C, зависит от значения A и не зависит от значения C.

FD является частным случаем MVD, когда множество значений зависимого атрибута обязательно состоит из одного элемента. Таким образом, если выполняется FD $A \rightarrow B$, то выполняется и MVD $A \twoheadrightarrow B$.

В переменной отношения R с атрибутами (возможно, составными) A и B MVD $A \twoheadrightarrow B$ называется *тривиальной*, если либо $A \subseteq B$, либо A UNION B совпадает с заголовком отношения R.

Тривиальная MVD всегда удовлетворяется. При $A \subseteq B$ она вырождается в тривиальную FD. В случае A UNION B = H_R требования многозначной зависимости соблюдаются очевидным образом.

Теорема Фейджина.

Лемма Фейджина

В отношении R {A, B, C} выполняется MVD $A \twoheadrightarrow B$ в том и только в том случае, когда выполняется MVD $A \twoheadrightarrow C$.Таким образом, MVD $A \twoheadrightarrow B$ и $A \twoheadrightarrow C$ всегда составляют пару. Поэтому обычно их представляют вместе в форме A $\twoheadrightarrow$ B | C.

Доказательство достаточности условия леммы. Пусть выполняется MVD $A \twoheadrightarrow B$. Пусть имеется некоторое удовлетворяющее этой зависимости значение r переменной отношения R, a обозначает значение атрибута A в некотором кортеже тела B_r, a {b} – множество значений атрибута B, взятых из всех кортежей B_r, в которых значением атрибута A является a. Предположим, что для этого значения a MVD $A \twoheadrightarrow C$ не выполняется. Это означает, что существуют такое допустимое значение c атрибута C и такое значение $b \in \{b\}$, что кортеж {a, b, c} $\notin B_r$. Но это противоречит наличию MVD $A \twoheadrightarrow B$. Следовательно, если выполняется MVD $A \twoheadrightarrow B$, то выполняется и MVD $A \twoheadrightarrow C$. Аналогично можно доказать необходимость условия леммы.

Теорема Фейджина

Пусть имеется переменная отношения R с атрибутами A, B, C (в общем случае, составными).

Отношение *R* декомпозируется без потерь на проекции {A, B} и {A, C} тогда и только тогда, когда для него выполняется MVD A $\rightarrow\!\!\!\!\rightarrow$ B | C.

Достаточность условия теоремы. Пусть *r* является некоторым допустимым значением переменной отношений *R*. Пусть a есть значение атрибута A в некотором кортеже тела B_r, {b} – множество значений атрибута B, взятых из всех кортежей тела B_r, в которых значением атрибута A является a, и {c} – множество значений атрибута C, взятых из всех кортежей тела B_r, в которых значением атрибута A является a. Тогда очевидно, что в тело значения *r* PROJECT {A, B} будут входить все кортежи вида {a, b_i}, где $b_i \in$ {b}, и если некоторый кортеж {a, b_j} входит в тело значения отношения *r* PROJECT {A, B}, то $b_j \in$ {b}. Аналогичные рассуждения применимы к *r* PROJECT {A, C}. Очевидно, что из этого следует, что при наличии многозначной зависимости A $\rightarrow\!\!\!\!\rightarrow$ B | C в переменной отношения *R*{A, B, C} декомпозиция *r* на проекции *r* PROJECT {A, B} и *r* PROJECT {A, C} является декомпозицией без потерь.

Необходимость условия теоремы. Предположим, что декомпозиция переменной отношения *R* {A, B, C} на проекции *R* PROJECT {A, B} и *R* PROJECT {A, C} является декомпозицией без потерь для любого допустимого значения *r* переменной отношения *R*. Мы должны показать, что в теле B_r значения-отношения *r* поддерживается ограничение

IF ({a, b_1, c_1}; $\in B_r$ AND {a, b_2, c_2} $\in B_r$) THEN ({a, b_1, c_2} $\in B_r$ AND {a, b_2, c_1} $\in B_r$)

Действительно, пусть в B_r входят кортежи {a, b_1, c_1} и {a, b_2, c_2}. Предположим, что {a, b_1, c_2} $\notin B_r$ OR a, b_2, $c_1 \notin B_r$. Но в тело значения отношения *r* PROJECT {A, B} входят кортежи {a, b_1} и {a, b_2}, а в тело значения переменной отношения *r* PROJECT {A, C} – {a, c_1} и {a, c_2};. Очевидно, что в тело значения естественного соединения *r* PROJECT {A, B} NATURAL JOIN *r* PROJECT {A, C} войдут кортежи {a, b_1, c_2} и {a, b_2, c_1}, и наше предположение об отсутствии по крайней мере одного из этих кортежей в B_r противоречит исходному предположению о том, что декомпозиция *r* на проекции *r* PROJECT {A, B} и *r* PROJECT {A, C} является декомпозицией без потерь.

Четвертая нормальная форма.

BCNF + все MVD r являются FD с детерминантами – возможными ключами отношения r.

В сущности, 4NF является BCNF, в которой многозначные зависимости вырождаются в функциональные.

(26)

Зависимости проекции-соединения.

n-декомпозируемое отношение – отношение, которое может быть декомпозировано без потерь на *n* проекций.

Пусть задана переменная отношения *R*, и A, B, …, Z являются произвольными подмножествами заголовка *R* (составными, перекрывающимися атрибутами). В переменной отношения *R* удовлетворяется *зависимость проекции/соединения (Project-Join Dependency – PJD)**(A, B, …, Z) тогда и только тогда, когда любое допустимое значение *r* переменной отношения *R* можно получить путем естественного соединения проекций этого значения на атрибуты A, B, …, Z.

В переменной отношения *R* PJD *(A, B, …, Z) называется *подразумеваемой возможными ключами* в том и только в том случае, когда каждый составной атрибут A, B, …, Z является

суперключом *R*, т. е. включает хотя бы один возможный ключ *R*.

В переменной отношения *R* зависимость проекции/соединения *(A, B, …, Z) называется *тривиальной*, если хотя бы один из составных атрибутов A, B, …, Z совпадает с заголовком *R*.

Пятая нормальная форма.

Переменная отношения *R* находится в *пятой нормальной форме*, или в *нормальной форме проекции/соединения* (5NF, или PJ/NF – Project-Join Normal Form) в том и только в том случае, когда каждая нетривиальная PJD в *R* подразумевается возможными ключами *R*.

(27)

Семантические модели данных.

Потребность проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области вызвала к жизни направление *семантических моделей данных*. Хотя 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.

Чаще всего 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. Основным достоинством данного подхода является отсутствие потребности в дополнительных программных средствах, поддерживающих семантическое моделирование. Требуется только знание основ выбранной семантической модели и правил преобразования концептуальной схемы в реляционную схему.

(28)

Семантическая модель Entity-Relationship (Сущность-Связи).

Основными понятиями ER-модели являются *сущность*, *связь* и *атрибут*.

Сущность – это реальный или представляемый объект, информация о котором должна сохраняться и быть доступной. В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности – это имя типа, а не некоторого конкретного экземпляра этого типа. При определении типа сущности необходимо гарантировать, что каждый экземпляр сущности может быть отличим от любого другого экземпляра той же сущности.

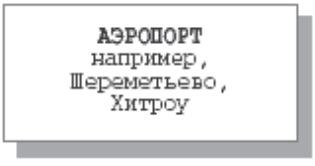

Рис. 10.1. Пример типа сущности

Связь – это графически изображаемая ассоциация, устанавливаемая между двумя типами сущностей. Как и сущность, связь – это типовое понятие, все экземпляры обоих связываемых типов сущностей подчиняются устанавливаемым правилам связывания. Поэтому правильнее говорить о типе связи, устанавливаемой между типами сущности, и об экземплярах типа связи, устанавливаемых между экземплярами типа сущности. В обсуждаемом здесь варианте ER-модели

эта ассоциация всегда является бинарной и может существовать между двумя разными типами сущностей или между типом сущности и им же самим (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указываются имя конца связи, степень конца связи (сколько экземпляров данного типа сущности должно присутствовать в каждом экземпляре данного типа связи), обязательность связи (т. е. любой ли экземпляр данного типа сущности должен участвовать в некотором экземпляре данного типа связи).

Связь представляется в виде ненаправленной линии, соединяющей две сущности или ведущей от сущности к ней же самой. При этом в месте «стыковки» связи с сущностью используются:

- трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут (или должны) использоваться много экземпляров сущности;
- одноточечный вход, если в связи может (или должен) участвовать только один экземпляр сущности.

Обязательный конец связи изображается сплошной линией, а необязательный – прерывистой линией.

Рис. 10.2. Пример типа связи

- каждый БИЛЕТ предназначен для одного и только одного ПАССАЖИРА;
- каждый ПАССАЖИР может иметь один или более БИЛЕТОВ.

Атрибутом сущности является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Имена атрибутов заносятся в прямоугольник, изображающий сущность, под именем сущности и изображаются малыми буквами, возможно, с примерами.

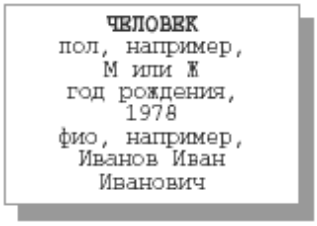

Рис. 10.4. Пример типа сущности с атрибутами

(29)

Получение реляционной схемы из ER-диаграммы.

Подготовка

В первой нормальной форме ER-диаграммы устраняются атрибуты, содержащие множественные значения, т. е. производится выявление неявных сущностей, «замаскированных» под атрибуты.

Рис. 10.9. Пример приведения ER-диаграммы к первой нормальной форме

Во второй нормальной форме устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность.

Рис. 10.10. Пример приведения ER-диаграммы ко второй нормальной форме

В третьей нормальной форме устраняются атрибуты, которые зависят от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности.

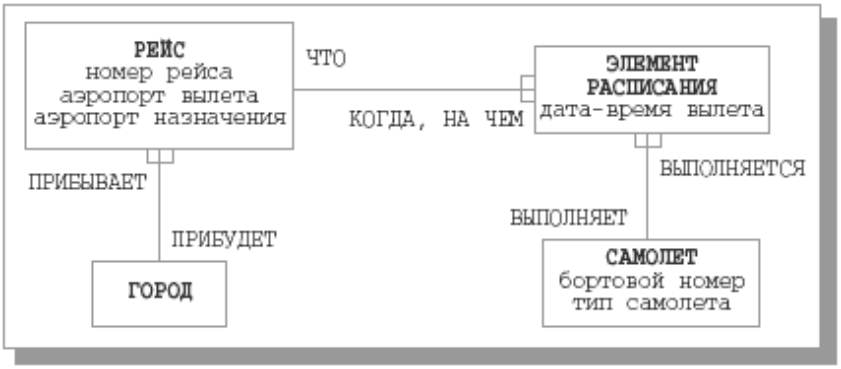

Рис. 10.11. Пример приведения ER-диаграммы к третьей нормальной форме

Получение реляционной схемы

Каждый простой тип сущности превращается в таблицу. (Простым типом сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов.) Имя сущности становится именем таблицы. Экземплярам типа сущности соответствуют *строки* соответствующей таблицы.

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

Компоненты уникального идентификатора сущности превращаются в *первичный ключ таблицы*. Если имеется несколько возможных уникальных идентификаторов, для первичного ключа выбирается наиболее характерный. 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, и в общем случае может привести к зацикливанию). Для именования этих столбцов используются имена концов связей и/или имена парных типов сущностей.

Связи «многие к одному» (и «один к одному») становятся внешними ключами, т. е. образуется копия уникального идентификатора сущности на конце связи «один», и соответствующие столбцы составляют внешний ключ таблицы, соответствующей типу сущности на конце связи «многие». Необязательные связи соответствуют столбцам внешнего ключа, допускающим наличие неопределенных значений; обязательные связи – столбцам, не допускающим неопределенных значений. Если между двумя типами сущности `A` и `B` имеется связь «один к одному», то соответствующий внешний ключ по желанию проектировщика может быть объявлен как в таблице `A`, так и в таблице `B`. Чтобы отразить в определении таблицы ограничение, которое заключается в том, что степень конца связи должна равняться единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы (в случае использования языка SQL для этого служит спецификация `UNIQUE` – см. лекцию 16).

Для поддержки связи «многие ко многим» между типами сущности `A` и `B` создается дополнительная таблица `AB` с двумя столбцами, один из которых содержит уникальные идентификаторы экземпляров сущности `A`, а другой – уникальные идентификаторы экземпляров сущности `B`. Обозначим через `УИД(c)` уникальный идентификатор экземпляра `c` некоторого типа сущности `C`. Тогда, если в экземпляре связи «многие ко многим» участвуют экземпляры a_1, a_2, ..., a_n типа сущности `A` и экземпляры b_1, b_2, ..., b_m типа сущности `B`, то в таблице `AB` должны присутствовать все строки вида {УИД(a_i), УИД(b_j)} для $i = 1, 2, ..., nn$, $j = 1, 2, ..., m$. Понятно, что, используя таблицы `A`, `B` и `AB`, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи.

Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов,

на которых предполагается в основном базировать запросы.

(30)

Диаграммы классов языка UML.

Диаграммой классов в терминологии UML называется диаграмма, на которой показан набор классов (и некоторых других сущностей, не имеющих явного отношения к проектированию БД), а также связей между этими классами. Кроме того, диаграмма классов может включать комментарии и ограничения. Ограничения могут неформально задаваться на естественном языке или же могут формулироваться на языке объектных ограничений OCL (Object Constraints Language).

Определение классов

Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически класс изображается в виде прямоугольника. У каждого класса должно быть имя (текстовая строка), уникально отличающее его от всех других классов.

Рис. 11.1. Примеры описания классов

Атрибутом класса называется именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов (в частности, не иметь ни одного атрибута). Свойство, выражаемое атрибутом, является свойством моделируемой сущности, общим для всех объектов данного класса. Так что атрибут является абстракцией состояния объекта. Любой атрибут любого объекта класса должен иметь некоторое значение.

Рис. 11.2. Класс Человек с указанными именами атрибутов

Операцией класса называется именованная услуга, которую можно запросить у любого объекта этого класса. Операция – это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

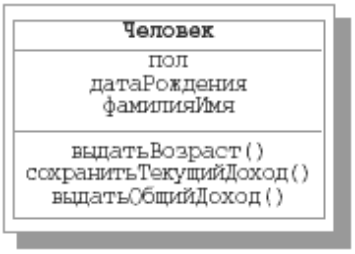

Рис. 11.3. Класс Человек с операциями

Определение связей

В диаграмме классов могут участвовать связи трех разных категорий: *зависимость* (dependency), *обобщение* (generalization) и *ассоциация* (association). При проектировании реляционных БД наиболее важны вторая и третья категории связей, поэтому о связях-зависимостях будет сказано только самое основное.

Зависимостью называют связь по применению, когда изменение в спецификации одного класса может повлиять на поведение другого класса, использующего первый класс. Чаще всего зависимости применяют в диаграммах классов, чтобы отразить в сигнатуре операции одного класса тот факт, что параметром этой операции могут быть объекты другого класса. Понятно, что если интерфейс второго класса изменяется, это влияет на поведение объектов первого класса.

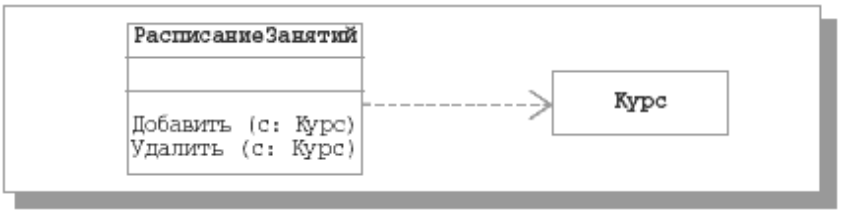

Рис. 11.4. Диаграмма классов со связью-зависимостью

Связью-обобщением называется связь между общей сущностью, называемой *суперклассом*, или родителем, и более специализированной разновидностью этой сущности, называемой *подклассом*, или потомком. Обобщения иногда называют *связями «is a»*, имея в виду, что класс-потомок является частным случаем класса-предка. Класс-потомок наследует все атрибуты и операции класса-предка, но в нем могут быть определены дополнительные атрибуты и операции.

Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют *полиморфизмом по включению*, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка.

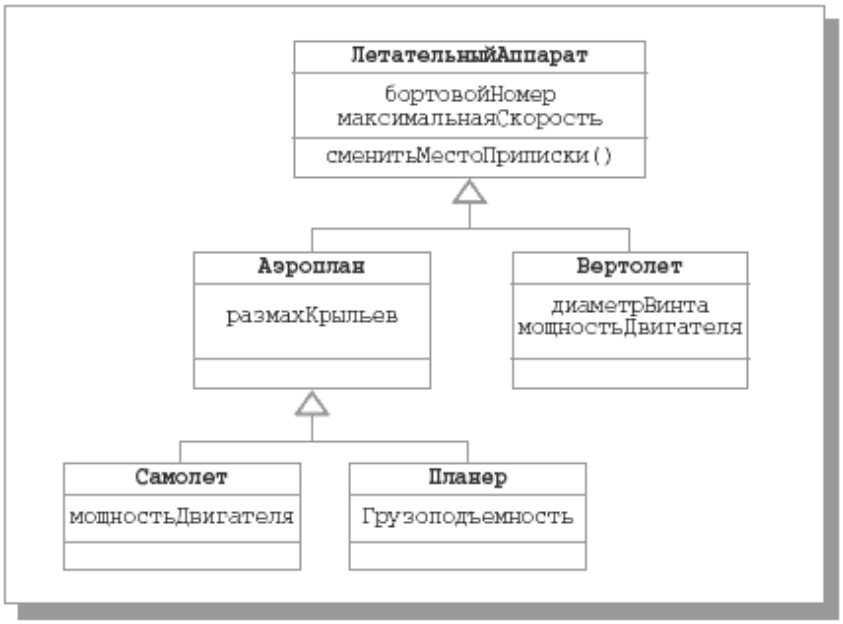

Рис. 11.5. Иерархия одиночного наследования классов

Одиночное наследование является достаточным в большинстве случаев применения связи-обобщения. Однако в UML допускается и *множественное наследование*, когда один подкласс

определяется на основе нескольких суперклассов.

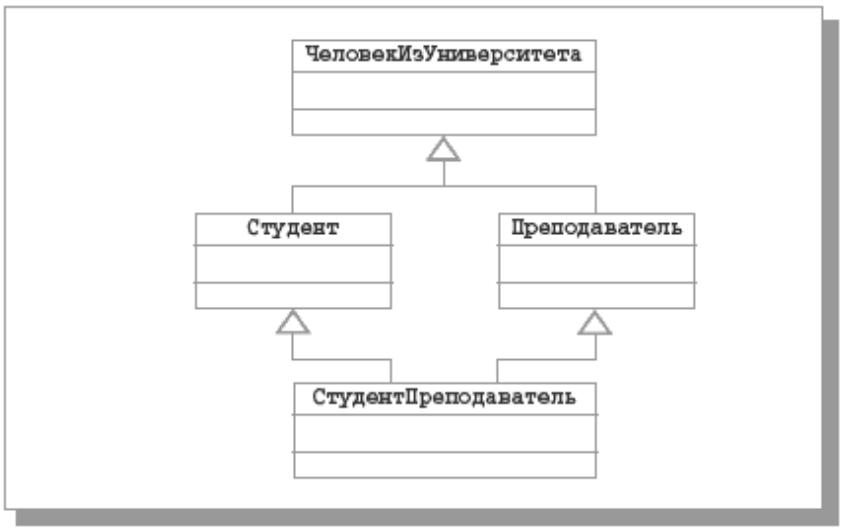

Рис. 11.6. Пример множественного наследования классов

Ассоциацией называется структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса. Допускается, чтобы оба конца ассоциации относились к одному классу. В ассоциации могут связываться два класса, и тогда она называется бинарной. Допускается создание ассоциаций, связывающих сразу n классов (они называются *n-арными ассоциациями*).

С понятием ассоциации связаны четыре важных дополнительных понятия: имя, роль, кратность и агрегация. Во-первых, ассоциации может быть присвоено имя, характеризующее природу связи. Смысл имени уточняется с помощью черного треугольника, который располагается над линией связи справа или слева от имени ассоциации. Этот треугольник указывает направление чтения имя связи.

Рис. 11.7. Пример именованной ассоциации

Другим способом именования ассоциации является указание роли каждого класса, участвующего в этой ассоциации. Роль класса, как и имя конца связи в ER-модели, задается именем, помещаемым под линией ассоциации ближе к данному классу.

Рис. 11.8. Две ассоциации с разными ролями классов

В общем случае, для ассоциации могут задаваться и ее собственное имя, и имена ролей классов. Это связано с тем, что класс может играть одну и ту же роль в разных ассоциациях, так что в общем

случае пара имен ролей классов не идентифицирует ассоциацию. С другой стороны, в простых случаях, когда между двумя классами определяется только одна ассоциация, можно вообще не связывать с ней дополнительные имена.

Кратностью (multiplicity) роли ассоциации называется характеристика, указывающая, сколько объектов класса с данной ролью может или должно участвовать в каждом экземпляре.

Рис. 11.9. Ассоциации с указанными кратностями ролей

Обычная ассоциация между двумя классами характеризует связь между равноправными сущностями: оба класса находятся на одном концептуальном уровне. Но иногда в диаграмме классов требуется отразить тот факт, что ассоциация между двумя классами имеет специальный вид «часть-целое». В этом случае класс «целое» имеет более высокий концептуальный уровень, чем класс «часть». Ассоциация такого рода называется *агрегатной*.

Рис. 11.10. Пример агрегатной ассоциации

Бывают случаи, когда связь «части» и «целого» настолько сильна, что уничтожение «целого» приводит к уничтожению всех его «частей». Агрегатные ассоциации, обладающие таким свойством, называются *композитными*, или просто композициями. При наличии композиции объект-часть может быть частью только одного объекта-целого (композита). При обычной агрегатной ассоциации «часть» может одновременно принадлежать нескольким «целым».

Рис. 11.11. Пример композитной агрегатной ассоциации

Заметим, что в контексте проектирования реляционных БД агрегатные и в особенности композитные ассоциации влияют только на способ поддержки ссылочной целостности. В частности, композитная связь является явным указанием на то, что ссылочная целостность между «целым» и «частями» должна поддерживаться путем каскадного удаления частей при удалении целого.

Навигация по ассоциации может проводиться в обоих направлениях.Однако бывают случаи, когда желательно ограничить направление навигации для некоторых ассоциаций.

Рис. 11.12. Ассоциация с указанным направлением навигации

(31)

Язык объектных ограничений OCL.

Язык OCL предназначен, главным образом, для определения ограничений целостности данных, соответствующих модели, которая представлена в терминах диаграммы классов UML.

Инвариант класса – это логическое выражение, вычисление которого должно давать `true` при создании любого объекта данного класса и сохранять истинное значение в течение всего времени существования этого объекта.

context <class_name> inv:
`<OCL-выражение>`

В общем случае OCL-выражение в определении инварианта основывается на композиции операций, которым посвящена большая часть определения языка. В спецификации языка эти операции условно разделены на следующие группы:

- операции над значениями предопределенных в UML (скалярных) типов данных;
- операции над объектами;
- операции над множествами;
- операции над мультимножествами;
- операции над последовательностями.

Операции над значениями скалярных типов

Поддерживаются следующие заимствованные из определения UML скалярные типы данных: Boolean, Integer, Real и String.

Операции над объектами

- получение значения атрибута–<объект>.<имя атрибута>,
- переход по соединению – <объект>.<имя роли, противоположенной по отношению к объекту>,
- вызов операции класса (последняя операция для целей проектирования реляционных БД несущественна).

Операции над множествами

<коллекция> → <имя операции: select, collect, exists, forAll, sizeб union, intersect, symmetricDifference> (<список фактических параметров>)

Примеры:

context Служащий inv: self.возраст > 18 and self.возраст < 100

context Отдел inv: self.номер ≤ 5 orself.служащий →select (возраст ≤ 30) →size () = 0

context Компания inv: self.отдел →collect (служащие) →size ( ) < 1000

(32)

Основные цели System R и их связь с архитектурой системы.

При выполнении проекта System R преследовались следующие основные цели:

- обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
- обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
- поддерживать динамически изменяемую среду баз данных, в которой таблицы, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
- обеспечить возможность параллельной работы с одной базой данных многих пользователей, допуская параллельную модификацию объектов базы данных при наличии необходимых средств защиты целостности базы данных;
- обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
- обеспечить гибкий механизм, позволяющий определять различные представления хранимых данных и ограничивать этими представлениями доступ пользователей к базе данных по выборке и модификации на основе механизма авторизации;
- обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Язык

Основой System R является «реляционный» язык SEQUEL (который достаточно быстро был переименован в SQL).

Целостность

Под целостным состоянием базы данных понимается состояние, удовлетворяющее набору сохраняемых при базе данных предикатов целостности. Эти предикаты, называемые в System R *утверждениями целостности* (assertion), также задаются средствами языка SQL. Любой оператор языка выполняется в границах некоторой *транзакции* – последовательности операторов языка, неделимой в смысле состояния базы данных. Неделимость означает, что все изменения базы данных, произведенные в пределах одной транзакции, либо целиком отображаются в состоянии базы данных, либо полностью в нем отсутствуют. Последняя возможность возникает при *откате* транзакции, который может произойти по инициативе пользователя (при выполнении соответствующего оператора SQL) или по инициативе системы.

Одной из причин отката транзакции по инициативе системы является как раз нарушение целостности базы данных в результате действий данной транзакции (другие возможные условия отката транзакции по инициативе системы мы рассмотрим позже). Язык SQL System R (так мы будем называть вариант языка SQL, разработанный в проекте System R, чтобы отличать его от более поздних, «стандартных» вариантов этого языка) содержит средство установки так называемых *точек сохранения* (savepoint). При инициируемом пользователем откате транзакции можно указать номер точки сохранения, выше которого откат не распространяется. Инициируемый системой откат транзакции производится до ближайшей точки сохранения, в которой условие, вызвавшее откат, уже отсутствует. В частности, откат транзакции, инициированный по причине нарушения условия целостности, производится до ближайшей точки сохранения, в которой условия целостности соблюдены.

Журналирование

Естественно, что для реального выполнения отката транзакции необходимо запоминать некоторую информацию о выполнении транзакции. В System R для этих и других целей используется специальный набор данных – *журнал*, в который помещаются записи обо всех операциях всех транзакций, изменяющих состояние базы данных. При откате транзакции происходит процесс *обратного выполнения* транзакции (undo), в ходе которого в обратном порядке выполняются все изменения, запомненные в журнале.

Ограничения доступа

В языке SQL System R имеется средство определения так называемых *триггеров* (trigger), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. В SQL System R триггер – это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения. Особенно существенно наличие такого механизма в связи с наличием обсуждаемых ниже представлений базы данных, которыми может быть ограничен доступ к базе данных для ряда пользователей. Возможна ситуация, когда такие пользователи просто не могут соблюдать целостность базы данных без автоматического выполнения условных воздействий, поскольку они просто «не видят» всей базы данных и, в частности, не могут представить всех ограничений ее целостности.

Параллельная работа

Основной подход System R состоит в том, что пользователь не обязан знать о наличии других пользователей, конкурирующих с ним за доступ к базе данных, т.е. система ответственна за обеспечение изолированности пользователей с гарантией отсутствия их взаимного влияния в пределах транзакций. Из этого следует, во-первых, что в интерфейсе пользователя с системой (т.е. в языке SQL) не должно быть средств регулирования взаимодействий с другими пользователями и, во-вторых, что система должна обеспечить автоматическую сериализацию набора транзакций, т.е. обеспечить режим выполнения этого набора транзакций, эквивалентный по конечному результату некоторому последовательному выполнению этих транзакций. Эта проблема решается в System R за счет автоматического выполнения синхронизационных блокировок всех изменяемых объектов базы данных.

Надежность

Одним из основных требований к СУБД вообще и к System R в частности является обеспечение надежности баз данных по отношению к различного рода сбоям. К таким сбоям могут относиться программные ошибки прикладного и системного уровня, сбои процессора, поломки внешних носителей и т.д. В частности, к одному из видов сбоев можно отнести упоминавшиеся выше нарушения целостности базы данных и автоматический инициируемый системой откат транзакции – это системное средство восстановления базы данных после сбоев такого рода. Как уже отмечалось, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. На информации журнала также основано восстановление базы данных и после сбоев другого рода.

Эффективность

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со спецификой физической организации баз данных во внешней памяти, использованием техники индексированного доступа к данным, буферизацией используемых страниц базы данных в основной памяти и развитой техникой оптимизации SQL-запросов, производимой на стадии их компиляции.

Структурная организация системы

Структурная организация System R согласуется с поставленными при ее разработке целями и

выбранными решениями. Основными структурными компонентами System R являются система управления реляционными данными (Relational Data System – RDS), состоящая, по существу, из компилятора языка SQL и подсистемы поддержки откомпилированных операторов, и система управления реляционной памятью (Relational Storage System – RSS).

RSS обеспечивает интерфейс довольно низкого, но достаточного для реализации SQL уровня для доступа к хранимым в базе данным. Синхронизация транзакций, журнализация изменений и восстановление баз данных после сбоев также относятся к числу функций RSS.

Компилятор запросов использует интерфейс RSS для доступа к разнообразной справочной информации (каталоги таблиц, индексов, прав доступа, условий целостности, условных воздействий и т.д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса RSS.

Таким образом, система естественно разделяется на два уровня – уровень управления памятью и синхронизацией, фактически, не зависящий от базового языка запросов системы, и языковый уровень (уровень SQL), на котором решается большинство проблем System R. Заметим, что эта независимость скорее условная, чем абсолютная: язык SQL можно в принципе заменить каким-либо другим языком, но он должен обладать примерно такой же семантикой.

(33)

Организация внешней памяти в базах данных System R.

Как уже говорилось, база данных System R располагается в одном или нескольких сегментах внешней памяти. Каждый сегмент состоит из страниц данных и страниц индексной информации. Размер страницы данных в сегменте может быть выбран равным либо 4, либо 32 килобайтам; размер страницы индексной информации равен 512 байтам. Кроме того, при работе RSS поддерживается дополнительный набор данных для ведения журнала. Для повышения надежности журнала (а это наиболее критичная информация; при ее потере восстановление базы данных после сбоев невозможно) этот набор данных дублируется на двух внешних носителях.

Страницы данных и идентификаторы кортежей

В каждой странице данных хранятся кортежи одной или нескольких таблиц. Фундаментальным понятием RSS является *идентификатор кортежа (tuple identifier – tid)*. Гарантируется неизменяемость tid'a во все время существования кортежа в базе данных независимо от перемещений кортежа внутри страницы и даже при перемещении кортежа в другую страницу. Потребность в перемещении кортежей возникает по той причине, что кортеж, занесенный в некоторую таблицу базы данных, вообще говоря, во время своего существования может увеличиваться в размерах (если к этой таблице добавляется новое поле, или если в ней имеется хотя бы одно поле, типом данных которого являются строки символов переменного размера). Реально tid представляет собой пару <номер страницы, индекс описателя кортежа в странице>. При этом кортеж может реально располагаться в данной странице или в другой странице.

Для динамического распределения памяти внутри страницы память на описатели кортежей выделяется вниз от начала страницы, а память для хранения кортежей – вверх от конца страницы.

Поскольку допускается нахождение в одной странице данных кортежей разных таблиц, каждый кортеж должен, кроме содержательной части, включать служебную информацию, идентифицирующую таблицу, которой принадлежит данный кортеж. Кроме того, в System R (точнее, в языке SQL) допускается динамическое добавление полей к существующим таблицам. При этом реально происходит лишь модификация описателя таблицы в таблице-каталоге таблиц. В существующем кортеже таблицы новое поле возникает только при модификации этого кортежа, затрагивающей новое поле. Это позволяет избежать массовой перестройки хранимой таблицы при

добавлении к ней новых полей, но, естественно, требует хранения при кортеже дополнительной служебной информации, определяющей реальное число полей в данном кортеже. (Заметим, что удалять существующие поля существующей таблицы в SQL System R не разрешалось.)

Индексы и кластеризация таблиц

Каждый *индекс* определяется на одном или нескольких полях таблицы, значения которых составляют его ключ, и позволяет производить прямой поиск по ключу кортежей (их tid'ов) и последовательное сканирование таблицы по индексу, начиная с указанного ключа, в порядке возрастания или убывания значений ключа. Некоторые индексы при их создании могут обладать атрибутом уникальности. В таком индексе не допускаются дубликаты ключа. Это единственное средство SQL System R указания системе первичного ключа таблицы (фактически, набора первичного и всех возможных ключей таблицы).

Для организации индексов в System R применяется техника *B+-деревьев*. Эту традицию соблюдает большинство реляционных систем, возникших позднее.

Видимо, наиболее важной особенностью физической организации баз данных в System R является возможность обеспечения *кластеризации* связанных кортежей одной или нескольких таблиц. Под кластеризацией кортежей понимается физически близкое расположение (в пределах одной страницы данных) логически связанных кортежей. Обеспечение соответствующей кластеризации позволяет добиться высокой эффективности системы при выполнении некоторого класса запросов.

В окончательном варианте System R существует только одно средство определения условий кластеризации таблицы – объявить до заполнения таблицы один (и только один) индекс, определенный на полях этой таблицы, кластеризованным. Тогда, если заполнение таблицы кортежами производится в порядке возрастания или убывания значений полей кластеризации (в зависимости от атрибутики индекса), система физически располагает кортежи в страницах данных в том же порядке.

Кроме того, в каждой странице данных кластеризованной таблицы оставляется некоторое резервное свободное пространство. При последующих вставках кортежей в такую таблицу система стремится поместить каждый кортеж в одну из страниц данных, в которых уже находятся кортежи этой таблицы с такими же (или близкими) значениями полей кластеризации. Естественно, что поддерживать идеальную кластеризацию таблицы можно только до определенного предела, пока не исчерпается резервная память в страницах. Далее этого предела степень кластеризации таблицы начинает уменьшаться, и для восстановления идеальной кластеризации таблицы требуется физическая реорганизация таблицы (ее можно произвести средствами SQL).

Списки

Список – это временная структура данных, создаваемая с целью оптимизации выполнения SQL-запроса, содержащая некоторые кортежи хранимой таблицы базы данных, не имеющая имени и, следовательно, не видимая на уровне интерфейса SQL. Кортежи списка могут быть упорядочены по возрастанию или убыванию полей соответствующей таблицы. Средства работы со списками имеются в интерфейсе RSS, но их, естественно, нет в SQL. Соответственно, эти средства используются только внутри системы при выполнении запросов (в частности, один из наиболее эффективных алгоритмов выполнения соединений основан на использовании отсортированных списков кортежей). Публикации по System R не дают точного представления о структурах данных, используемых при организации списков, но исходя из здравого смысла можно предположить, что они устроены не так, как таблицы (например, для кортежа, входящего в список, не требуется адресация через tid), и что располагаются они во временных файлах (в случае сбоя системы все временные объекты пропадают).

B-деревья.

Для организации индексов в System R применяется техника *B+-деревьев*. Каждый индекс занимает отдельный набор страниц, номер корневой страницы запоминается в описателе индекса. Использование B+-деревьев позволяет достичь эффективности при прямом поиске, поскольку они из-за своей *сильной ветвистости* обладают небольшой глубиной. Кроме того, B+-деревья сохраняют *порядок ключей* в листовых блоках иерархии, что позволяет производить последовательное сканирование таблицы в порядке возрастания или убывания значений полей, на которых определен индекс. Фундаментальное свойство B+-деревьев – *автоматическая балансировка* дерева – допускает произведение лишь локальных модификаций индекса при переполнениях и опустошениях страниц индекса.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). В B+-дереве внутренние и листовые страницы обычно имеют разную структуру.

Структура внутренней страницы:

- $ключ_1 \leq ключ_2 \leq \ldots \leq ключ_m$;
- в странице дерева N_m находятся ключи k со значениями $ключ_m <= k <= ключ_{m+1}$.

Структура листовой страницы:

- $ключ_1 < ключ_2 < \ldots < ключ_k$;
- $список_r$ – упорядоченный список идентификаторов кортежей (tid), включающих значение $ключ_r$;
- листовые страницы связаны одно- или двунаправленным списком.

Поиск в B+-дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. Заметим, что поскольку B+-деревья являются сильно ветвистыми и сбалансированными, 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 внутренней странице помещается n ключей, то при хранении m записей требуется дерево глубиной $\log_n(m)$. Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск. Основной «изюминкой» B+-деревьев является автоматическое поддержание свойства сбалансированности.

Хэширование

Альтернативным и достаточно популярным подходом к организации индексов является использование техники *хэширования*. Общей идеей является применение к значению ключа некоторой *функции свертки (хэш-функции)*, вырабатывающей значение меньшего размера. Значение хэш-функции 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 значение свертки (одним из распространенных видов «хороших» хэш-функций являются функции, выдающие остаток от деления значения ключа на некоторое простое число). 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, но возникнет слишком много цепочек переполнения, и 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).

(34)

Интерфейс ядра System R - RSS.

Вместо имен объектов используются их уникальные идентификаторы, являющиеся прямыми или косвенными адресами внутренних описателей объектов во внешней памяти для постоянных объектов или в основной памяти для временных объектов. Замена имен объектов базы данных на их идентификаторы производится компилятором SQL на основе информации, черпаемой им из системных таблиц-каталогов.

Группы операций:

- операции сканирования таблиц и списков;
- операции создания и уничтожения постоянных и временных объектов базы данных;
- операции модификации таблиц и списков;
- операция добавления поля к таблице;
- операции управления прохождением транзакций;
- операция явной синхронизации.

Операции сканирования таблиц и списков

Операции группы сканирования позволяют последовательно, в порядке, определяемом типом сканирования, прочитать кортежи таблицы или списка, удовлетворяющие требуемым условиям. Группа включает операции `OPEN`, `NEXT` и `CLOSE`, означающие, соответственно, начало сканирования, требование чтения следующего кортежа, удовлетворяющего условиям, и конец сканирования.

В результате успешного выполнения операции открытия сканирования (если нет ошибок в параметрах) вырабатывается и возвращается идентификатор сканирования, который используется в качестве аргумента других операций этой группы.

Семантика операции `NEXT` следующая: начиная с текущей позиции сканирования выбираются кортежи таблицы в порядке, определяемом типом сканирования, до тех пор, пока не встретится кортеж, значения полей которого удовлетворяют указанному условию. Этот кортеж и является результатом операции. Если при выборке кортежа достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании через индекс или последний кортеж таблицы или списка при прямом сканировании), то вырабатывается особый признак результата. После этого единственным разумным действием является закрытие сканирования – операция `CLOSE`

Операция `CLOSE` может быть выполнена в данной транзакции по отношению к любому ранее открытому сканированию независимо от его состояния (т.е. независимо от того, достигнута ли при сканировании правая граница диапазона сканирования). Параметром операции является идентификатор сканирования, и ее выполнение приводит к тому, что этот идентификатор становится недействительным (и, соответственно, уничтожаются служебные структуры памяти RSS, относящиеся к данному сканированию).

Операции создания и уничтожения постоянных и временных объектов базы данных

Группа операций создания и уничтожения постоянных и временных объектов базы данных включает операции создания таблиц (`CREATE TABLE`), списков (`CREATE LIST`), индексов (`CREATE IMAGE`) и уничтожения любого из подобных объектов (`DROP TABLE`, `DROP LIST` и `DROP IMAGE`). Входным параметром операций создания таблиц и списков является спецификатор структуры объекта, т.е. число полей объекта и спецификаторы их типов. Кроме того, при спецификации полей таблицы указывается разрешение или запрещение наличия неопределенных значений полей в кортежах этой таблицы или списка. Неопределенные значения кодируются специальным образом. Любая операция сравнения константы данного типа с неопределенным значением по определению вырабатывает значение *false*, кроме операции сравнения на совпадение со специальной литеральной

константой NULL.

В результате выполнения этих операций заводится описатель в служебной таблице описателей таблиц или основной памяти (в зависимости от того, создается ли постоянный объект или временный), и вырабатывается идентификатор объекта, который служит входным параметром других операций, относящихся к соответствующему объекту (в частности, параметром операции `OPEN` при открытии сканирования объекта).

Операции `DROP TABLE`, `DROP LIST` и `DROP IMAGE` могут быть выполнены в любой момент независимо от состояния объектов. Выполнение операции приводит к уничтожению соответствующего объекта и, вследствие этого, недействительности его идентификатора.

Операции модификации таблиц и списков

Группа операций модификации таблиц и списков включает операции вставки кортежа в таблицу или список (`INSERT`), удаления кортежа из таблицы (`DELETE`) и обновления кортежа в таблице (`UPDATE`).

В результате успешного выполнения операции вставки кортежа в таблицу вырабатывается идентификатор нового кортежа, который выдается в качестве результата операции и может быть в дальнейшем использован как прямой параметр операций удаления и модификации кортежей таблицы. При занесении кортежа в список значение идентификатора кортежа не вырабатывается (для списков допускается только последовательное сканирование и добавление новых кортежей в конец списка; над ними нельзя определить индексов, и поэтому косвенная адресация кортежей списков через их идентификаторы не требуется).

Кроме описанных «атомарных» операций сканирования и модификации таблиц и списков, интерфейс RSS включает одну «макрооперацию» `BUILDLIST`, позволяющую за одно обращение к RSS построить список, отсортированный в соответствии со значениями заданных полей. Эта операция включает сканирование заданной таблицы или списка, создание нового списка, в который включаются указанные поля выбираемых кортежей, и сортировку построенного списка в соответствии со значениями указанных полей. Идентификатор заново построенного отсортированного списка является ответным параметром операции.

Операция добавления поля к существующей таблице

Операция RSS добавления поля к существующей таблице позволяет в динамике изменять схему таблицы. Параметрами операции `CHANGE` являются идентификатор существующей таблицы и спецификация нового поля (его тип). При выполнении операции изменяется только описатель данной таблицы в служебной таблице описателей таблиц. До выполнения первой операции `UPDATE`, затрагивающей новое поле таблицы, реально ни в одном кортеже таблицы память под новое поле выделяться не будет. По умолчанию значения нового поля во всех кортежах таблицы, в которые еще не производилось явное занесение значения, считаются неопределенными. Тем самым, ни для одного поля, динамически добавленного к существующей таблице, не может быть запрещено хранение неопределенных значений.

Операции управления прохождением транзакций

Каждая операция RSS выполняется в пределах некоторой транзакции. Интерфейс RSS включает набор операций управления прохождением транзакции: начать транзакцию (`BEGIN TRANSACTION`), закончить транзакцию (`END TRANSACTION`), установить точку сохранения (`SAVE`) и выполнить откат до указанной точки сохранения или до начала транзакции (`RESTORE`).

При выполнении своих транзакций пользователи System R изолированы один от другого, т.е. не ощущают того, что система функционирует в многопользовательском режиме. Это достигается за счет наличия в RSS механизма неявной синхронизации. До конца транзакции никакие изменения базы данных, произведенные в пределах этой транзакции, не могут быть использованы в других

транзакциях (попытка использования таких данных приводит к временным синхронизационным блокировкам этих транзакций). При выполнении операции `END TRANSACTION` происходит "фиксация" изменений, произведенных в данной транзакции, т.е. они становятся видимыми в других транзакциях. Реально это означает снятие синхронизационных блокировок с объектов базы данных, изменявшихся в транзакции. Из этого следует, что после выполнения `END TRANSACTION` невозможны индивидуальные откаты данной транзакции. RSS просто делает недействительным идентификатор данной транзакции, и после выполнения операции окончания транзакции отвергает все операции с таким идентификатором.

Операция явной синхронизации

Последняя операция интерфейса RSS – операция явной синхронизации `LOCK`. Эта операция позволяет установить явную синхронизационную блокировку указанной таблицы (параметром операции является идентификатор таблицы). Выполнение операции `LOCK` гарантирует, что никакая другая транзакция до конца данной не сможет изменить эту таблицу (вставить в нее новый кортеж, удалить или модифицировать существующий), если установлена блокировка в режиме чтения, или даже прочитать любой кортеж этой таблицы, если установлена монопольная блокировка.

Но ситуации, в которых очевидна выгода от использования явной синхронизации, достаточно редки. Пользоваться этим средством можно только очень осмотрительно, потому что неоправданные захваты таких крупных объектов могут резко ограничить степень асинхронности выполнения транзакций.

(35)

ACID-транзакции.

ACID – Atomicy, Consistency, Isolation и Durability. В соответствии с этим понятием под транзакцией разумеется последовательность операций над базой данных, обладающая следующими свойствами.

- *Атомарность (Atomicy)*. Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных).
- *Согласованность (Consistency)*. В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций только в том случае, когда действия операций не нарушают *целостность* базы данных, т.е. удовлетворяют набору ограничений целостности, определенных для этой базы данных.
- *Изоляция (Isolation)*. Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую.
- *Долговечность (Durability)*. После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоев аппаратуры или программного обеспечения.

Средства СУБД для поддержки свойств атомарности, согласованности, изолированности и постоянства хранения.

Атомарность

При завершении транзакции оператором `COMMIT` (высокоуровневый аналог операции `END TRANSACTION` в интерфейсе RSS) результаты гарантированно фиксируются во внешней памяти (смысл термина *commit* состоит в запросе «фиксации» результатов транзакции); при завершении

транзакции оператором ROLLBACK (высокоуровневый аналог операции RESTORE в интерфейсе RSS) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Целостность

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 (например, возраст сотрудника не может превышать 150 лет). 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 с откатом транзакции до ее начала или до установленной ранее точки сохранения или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

Изолированность

Чтобы избежать ситуации *потерянных изменений* требуется, чтобы до завершения транзакции никакая другая транзакция не могла изменять никакой измененный текущей транзакцией объект (в частности, достаточно заблокировать доступ по изменению к этому объекту до завершения текущей транзакции).

Чтобы избежать ситуации *чтения "грязных" данных*, до завершения транзакции, изменившей объект базы данных, никакая другая транзакция не должна читать этот объект (например, достаточно заблокировать доступ по чтению к объекту до завершения изменившей его транзакции).

Чтобы избежать *неповторяющихся чтений*, до завершения транзакции никакая другая транзакция не должна изменять объект (для этого достаточно заблокировать доступ по записи к объекту до завершения транзакции чтения).

Чтобы избежать появления *кортежей-фантомов*, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

(36)

Сериализация 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.

Виды конфликтов транзакций и порождаемые ими феномены поведения транзакций.

- W/W –ситуация потерянных изменений;
- R/W – ситуация неповторяющихся чтений;
- W/R – ситуация «грязного» чтения.

Двухфазный протокол синхронизационных блокировок.

(Two-Phase Locking Protocol, 2PL). В общих чертах подход состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных o от имени транзакции T запрашивается синхронизационная блокировка объекта o в соответствующем режиме (в зависимости от вида операции).

В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции (выполнение операций над базой данных) – накопление блокировок;
- вторая фаза (фиксация или откат) – снятие блокировок.

Основными режимами синхронизационных блокировок являются следующие:

- совместный режим – S (Shared), означающий совместную (по чтению) блокировку объекта и требуемый для выполнения операции чтения объекта;
- монопольный режим – X (eXclusive), означающий монопольную (по записи) блокировку объекта и требуемый для выполнения операций вставки, удаления и модификации объекта.

(37)

Синхронизационные тупики.

- транзакции T_1 и T_2 устанавливают монопольные блокировки объектов o_1 и o_2 соответственно;
- после этого T_1 требуется совместная блокировка объекта o_2, а T_2 – совместная блокировка объекта o_1;
- ни одно из этих требований блокировки не может быть удовлетворено, следовательно, ни одна из транзакций не может продолжаться; поэтому монопольные блокировки объектов никогда не будут сняты, а требования совместных блокировок не будут удовлетворены.

Способы обнаружения тупиков.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций – это ориентированный двудольный граф, в котором существует два типа вершин – вершины, соответствующие транзакциям, и вершины, соответствующие объектам блокировок.

Прежде всего, из графа ожидания удаляются все дуги, исходящие из транзакций, в которые не входят дуги из объектов. (Это основывается на том разумном предположении, что транзакции, не ожидающие удовлетворения запроса блокировок, могут успешно завершиться и освободить блокировки). Кроме того, удаляются дуги, входящие в транзакции, из которых не исходят дуги к объектам (транзакции, ожидающие удовлетворения блокировок, но не удерживающие заблокированные объекты, не могут быть причиной тупика). Для тех объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация одной из исходящих дуг (выбираемой произвольным образом) изменяется на противоположную (это моделирует удовлетворение запроса блокировки). После этого снова повторяются описанные действия до тех пор, пока не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл.

Способы разрушения тупиков.

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. Обычно при выборе транзакции-жертвы используется многофакторная оценка ее стоимости, в которую с разными весами входят время выполнения, число накопленных блокировок, приоритет и т.д.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный (до некоторой точки сохранения) характер. При этом, естественно, освобождаются блокировки, и может быть продолжено выполнение других транзакций.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать. В централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

(38)

Гранулированные блокировки.

Перед установкой блокировки на некоторый объект базы данных в режиме S или X соответствующий объект верхнего уровня должен быть заблокирован в режиме IS, IX или SIX.

	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

Таблица 9.2. Совместимость блокировок S, X, IS, IX и SIX

Предикатные блокировки.

Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием блокировок таблиц в режимах S и X). Давно известно, что для решения этой проблемы необходимо перейти от блокировок индивидуальных («физических») объектов базы данных, к блокировке условий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для блокировок уровня таблиц именно потому, что таблица как логический объект представляет собой неявное условие для входящих в него кортежей. Блокировка таблицы – это простой и частный случай предикатной блокировки.

В System R блокировки сегментов (файлов), таблиц и кортежей технически трактовались единообразно, как блокировки идентификаторов кортежей (tid'ов). При блокировке кортежа на самом деле блокировался его tid. При блокировке сегмента или таблицы на самом деле блокировался tid описателя соответствующего объекта во внутренних таблицах-каталогах сегментов или таблиц.

При открытии сканирования всегда можно указать, для какой цели оно будет использоваться: для выборки кортежей, для их удаления или для их обновления (это известно компилятору SQL). Поэтому в случае типовой организации SQL-ориентированной СУБД простые условия можно использовать как основу предикатных захватов.

Чтобы гарантированно заблокировать таблицу целиком, достаточно заблокировать условие &1≤i≤n

(min(mi) < имя_поляi < max(mi)). Чтобы заблокировать базу данных, достаточно заблокировать условие, являющееся конъюнкцией условий блокировки всех таблиц этой базы данных.

(39)

Сериализация транзакций на основе временных меток.

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редкого возникновения конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании *временных меток*. Основная идея метода временных меток (Timestamp Ordering, TO), у которого существует множество разновидностей, состоит в следующем: если транзакция T_1 началась раньше транзакции T_2, то система обеспечивает такой сериальный план, как если бы транзакция T_1 была целиком выполнена до начала T_2.

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

Если T_1 «старше» T_2, то производится откат T_2, после чегоT_2 получает новую временную метку и начинается заново.

К недостаткам метода TO относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов, 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.

Версионный вариант алгоритма временных меток

Одним из наиболее старых и простых версионных алгоритмов является *версионный вариант алгоритма временных меток (Multiversion Timestamp Ordering, MVTO)*. Как и в простом методе временных меток, описанном в предыдущем подразделе, в алгоритме MVTO порядок выполнения операций одновременно выполняемых транзакций задается порядком временных меток, которые получают транзакции во время старта. Временные метки также используются для идентификации версий данных при чтении и модификации – каждая версия получает временную метку той транзакции, которая ее записала. Алгоритм MVTO не только следит за порядком выполнения операций транзакций, но также отвечает за трансформацию операций над объектами базы данных в операции над версиями этих объектов, т.е. каждая операция над объектом базы данных o преобразуется в соответствующую операцию над некоторой версией объекта o.

(40)

Ситуации, требующие восстановления 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 разрушении 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 основной памяти) и т.д. Ситуация характеризуется потерей той части базы данных, которая к моменту

сбоя содержалась в буферах оперативной памяти СУБД.

- 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.

Понятие журнала.

Во всех трех случаях основой восстановления является хранение избыточных данных. Эти избыточные данные хранятся в *журнале*, содержащем последовательность записей об изменении базы данных.

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 основной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

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

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 (т.е. должно поддерживаться свойство *долговечности (durability)* транзакций);
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных (в противном случае состояние базы данных могло бы оказаться не целостным).

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

Индивидуальные откаты транзакций.

Для обеспечения возможности индивидуального отката транзакции по общему журналу все записи в журнале от данной транзакции связываются в обратный список. В начале списка для незавершенных транзакций находится запись о последнем изменении базы данных, произведенном данной транзакцией. Заметим, что в этом случае хронологически последние записи могут быть еще не вытолкнуты во внешнюю память журнала и могут находиться в буфере основной памяти. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала, т.е. весь список находится во внешней памяти. 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 данной транзакцией.

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

- Выбирается очередная журнальная 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 транзакции, для которых не полностью выполнен индивидуальный откат.
- При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

Протокол Write Ahead Log.

Имеются два вида буферов – буфера журнала и буферный пул страниц основной памяти, – которые содержат связанную информацию. И те, и другие буфера могут выталкиваться во внешнюю память. Основной причиной выталкивания буфера журнала является его полное заполнение журнальными записями. Страницы буферного пула базы данных чаще всего выталкиваются во внешнюю память, когда требуется переместить в основную память некоторый блок базы данных, а свободных страниц в буферном пуле нет. Тогда срабатывает алгоритм замещения страниц, выбирается страница, содержимое которой, вероятно, дольше всего не потребуется, и эта страница (если ее содержимое изменялось) выталкивается в соответствующий блок внешней памяти базы данных. Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния базы данных после сбоев.

Заметим, что эта проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое основной памяти не утрачено, и при восстановлении можно пользоваться содержимым как буфера журнала, так и буферных страниц базы данных. Но если произошел мягкий сбой, и содержимое буферов утрачено, то 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 буферных страниц базы данных является то, что запись об изменении объекта базы данных должна оказаться во внешней памяти журнала раньше, чем измененный объект окажется во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется WAL (*Write Ahead Log*, «пиши сначала в журнал») и состоит в том, что если требуется вытолкнуть во внешнюю память буферную страницу, содержащую измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта.

При следовании протоколу WAL, 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.

(41)

Восстановление базы данных после мягкого сбоя.

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, блок данных и несколько блоков индексов. Блоки базы данных буферизуются в

оперативной памяти и выталкиваются независимо. После мягкого сбоя набор блоков внешней памяти базы данных может оказаться несогласованным, т.е. часть блоков внешней памяти соответствует объекту до изменения, часть – после изменения.

Физически согласованное состояние базы данных.

Точки *физической согласованности* базы данных – моменты времени, в которые 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 ppc (point of physical consistency).

Способы восстановления физически согласованного состояния.

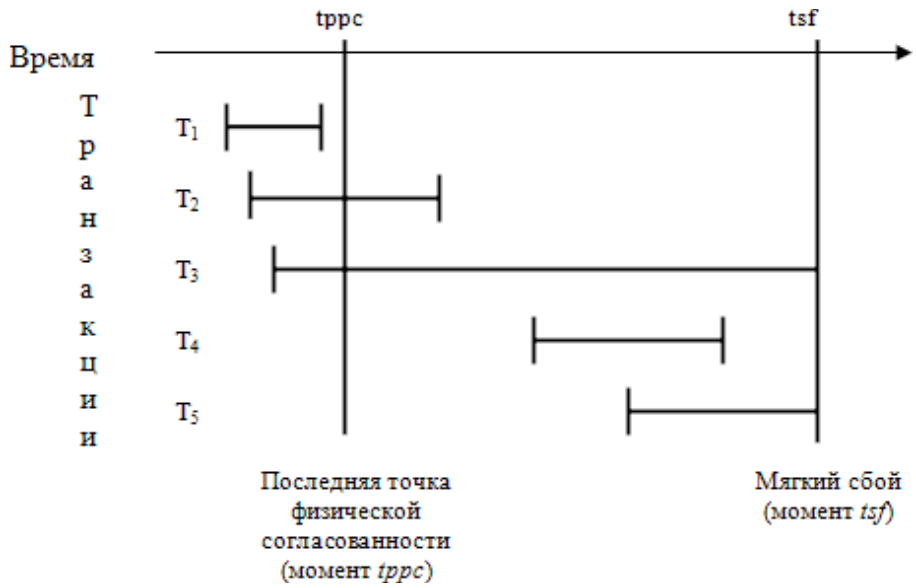

Рис. 14.1. Возможные состояния транзакций к моменту мягкого сбоя

Восстановление последнего по времени логически целостного состояния базы данных при условии восстановленного состояния на момент tpprспроизводится следующим образом.

- Для транзакции T_1 никаких действий производить не требуется. Она закончилась до момента *tppc*, и все ее результаты гарантированно отражены во внешней памяти базы данных.
- Для транзакции T_2 нужно повторно выполнить (*redo*) последовательность операций, которые выполнялись после установки точки физически согласованного состояния в момент *tppc*. Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T_2 после момента *tppc*. Следовательно, повторное прямое (по смыслу и хронологии) выполнение операций транзакции T_2 корректно и приведет к логически согласованному состоянию базы данных. (Поскольку транзакция T_2 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией.)
- Для транзакции T_3 нужно выполнить в обратном направлении (undo) ту часть операций, которую она успела выполнить до момента *tppc*. Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T_3, которые были выполнены после момента *tppc*. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T_3, которые были выполнены до момента *tppc*. Следовательно, обратное выполнение (по смыслу и хронологии) операций T_3 корректно и приведет к согласованному

состоянию базы данных. (Поскольку транзакция T_3 не завершилась к моменту мягкого сбоя *tfs*, при восстановлении необходимо устранить все последствия ее выполнения.)

- Для транзакции T_4, которая успела начаться после момента *tppc* и закончиться до момента мягкого сбоя *tfs*, нужно произвести полное повторное выполнение операций в прямом направлении. (Поскольку транзакция T_4 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией).
- Наконец, для транзакции T_5, начавшейся после момента *tppc* и не успевшей завершиться к моменту мягкого сбоя *tfs*, никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

Как восстановить состояние базы данных в момент *tppc*? Для этого используются два основных подхода: подход, основанный на использовании *теневого механизма*, и подход, в котором применяется *журнализация постраничных изменений* базы данных.

Теневой механизм

Теневой механизм был изначально предложен для поддержания целостности файлов при аварийном отключении питания компьютера. Файл представляется как набор блоков внешней памяти, для доступа к которым поддерживается таблица отображения. 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 основной памяти) изменяется, а теневая остается 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 основную память теневую таблицу отображения.

Здесь имеется некоторая проблема, состоящая в том, что в любой момент времени теневая таблица отображения должна быть корректной, т.е. соответствовать некоторому ранее зафиксированному физически целостному состоянию базы данных. Для этого необходимо обеспечить атомарность операции замены теневой таблицы отображения.

Журнализация постраничных изменений

Возможен другой подход, при использовании которого наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате недовыполненных 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. Вообще, выполнение логических операций уровня RSS носит транзакционный характер. В частности, как уже отмечалось выше, при выполнении логической операции обновления базы данных, вообще говоря, изменяется несколько блоков базы данных. Для обеспечения возможности отката отдельной операции (а это может потребоваться, например, если обнаруживается нарушение свойства уникальности какого-либо индекса) приходится до конца операции монопольно блокировать все страницы буферного пула базы данных, содержащие копии изменяемых этой операцией блоков базы данных.

Чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, при выталкивании любой страницы из буферного пула основной памяти в нее помещается номер последней записи о постраничном изменении этой страницы. Этот же номер запоминается в самой записи. Тогда, чтобы понять, нужно ли применить данную запись о постраничном изменении соответствующего блока внешней памяти для восстановления состояния этого блока, требуется всего лишь сравнить номер, содержащийся в этом блоке, с номером, содержащимся в журнальной

записи. Если в блоке содержится номер, меньший номера журнальной записи, то это означает, что буферная страница, в которой выполнялось соответствующее изменение, не была к моменту мягкого сбоя вытолкнута во внешнюю память, и применять данную запись для восстановления соответствующего блока внешней памяти не требуется.

(42)

Архивация базы данных и журнала.

Самый простой способ состоит в архивировании базы данных по явному указанию администратора или при переполнении журнала. Но можно выполнять архивацию базы данных реже, чем переполняется журнал. Вместо базы данных можно архивировать сам журнал. В пределе для полного восстановления базы данных после жесткого сбоя достаточно иметь исходную архивную копию базы данных, последовательность архивных копий журналов и последний логический журнал.

Может показаться, что восстановление базы данных на основе таких архивных источников будет занимать недопустимо большое время, однако здесь возможна значительная оптимизация в связи с тем, что архивированный логический журнал можно сжимать. Для этого для каждого объекта базы данных нужно найти последовательность журнальных записей, относящихся к этому объекту, в хронологическом порядке и заменить их одной записью, соответствующей операции над объектом, результат которой эквивалентен результату последовательного выполнения журнализованных операций из построенной последовательности.

Восстановление базы данных после жесткого сбоя.

Восстановление начинается с обратного копирования (на исправный носитель) 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.

(43)

Управление буферами основной памяти.

Журнализация операций изменения базы данных тесно связана не только с управлением транзакциями, но и с буферизацией блоков базы данных в основной памяти. По причинам объективно существующей разницы в скорости работы процессоров и основной памяти и устройств внешней памяти (эта разница в скорости существовала, существует, и будет существовать всегда) буферизация блоков базы данных в основной памяти является единственным реальным способом достижения приемлемой эффективности СУБД. Без поддержки буферизации базы данных СУБД работала бы со скоростью магнитных дисков, т.е. на несколько порядков медленнее, чем если бы обработка данных происходила в основной памяти.

Следует заметить, что здесь идет речь об использовании буферов (и базы данных, и журнала), располагающихся именно в физической основной памяти, управляемой непосредственно СУБД, а не виртуальной памяти СУБД, управляемой операционной системой.