ДИСЦИПЛИНА

"Базы данных"

**Вопросы к экзамену**

Оглавление

[1. Сферы применения баз данных и СУБД. 2](#_Toc503717074)

[2. Базы данных и файловые системы. Области применения файлов. Потребности информационных систем. 4](#_Toc503717075)

[3. Основные функции СУБД. Типовая организация современной СУБД. 13](#_Toc503717076)

[4. Ранние подходы к организации БД. 18](#_Toc503717077)

[5. Реляционный подход к организации БД. Базовые понятия реляционных баз данных. 20](#_Toc503717078)

[6. Фундаментальные свойства отношений. Реляционная модель данных. 25](#_Toc503717079)

[7. Базисные средства манипулирования реляционными данными. Реляционная алгебра. Реляционное исчисление. 28](#_Toc503717080)

[8. Проектирование реляционных БД. Нормализация. 33](#_Toc503717081)

[9. Основные понятия модели «Сущность-Связь». ER-диаграммы. 40](#_Toc503717082)

[10. Система System R. Основные цели System R. 49](#_Toc503717083)

[11. Структуры внешней памяти, методы организации индексов. Хранение отношений. 54](#_Toc503717084)

[12. Транзакции и целостность баз данных. Сериализация транзакций. 56](#_Toc503717085)

[13. Изолированность пользователей. 65](#_Toc503717086)

[14. Язык SQL, функции и основные возможности. Стандартизация SQL. 68](#_Toc503717087)

[15. Журнализация изменений БД. Журнализация и буферизация. 79](#_Toc503717088)

[16. Восстановление баз данных в различных ситуациях. 82](#_Toc503717089)

[17. Язык SQL, функции и основные возможности. Стандартизация SQL. 85](#_Toc503717090)

[18. Типы данных языка SQL. Средства определения схемы. 86](#_Toc503717091)

[19. Язык SQL, средства манипулирования данными. Структура запросов. 95](#_Toc503717092)

[20. Использование SQL при прикладном программировании. 99](#_Toc503717093)

[21. Проблемы оптимизации. Схема обработки запроса. 101](#_Toc503717094)

[22. Архитектура "клиент-сервер". Серверы баз данных. 107](#_Toc503717095)

[23. Распределенные БД. 111](#_Toc503717096)

[24. Системы управления базами данных следующего поколения. 116](#_Toc503717097)

[25. Объектно-ориентированные СУБД. 124](#_Toc503717098)

[26. Системы баз данных, основанные на правилах. Активные и дедуктивные базы данных. 141](#_Toc503717099)

## 1. Сферы применения баз данных и СУБД.

База данных помогает систематизировать и хранить информацию из определенной предметной области, облегчает доступ к данным, поиск и предоставление необходимых сведений.

|  |
| --- |
| Одной из важных сфер компьютеризации общества является автоматизация управленческо-производственной, конторской и другой учрежденческой работы, где необходимы сбор, хранение, переработка, получение, передача информации, помощь в принятии решений. Автоматизация на основе персональных компьютеров ПК меняет стереотипы обработки информации, позволяет организовать функционирование предприятий и организаций на базе новой информационной технологии. Использование ПК в качестве инструмента обработки данных в различных областях человеческой деятельности повышает информационную культуру общества, способствует безболезненному переходу к информационному обществу, где информация является важнейшим стратегическим сырьем наравне с материальными, энергетическими, людскими и др. ресурсами. От вовремя полученной, правильно обработанной и четко представленной информации часто зависит эффективность принимаемых решений и их результат. Если постараться классифицировать существующие области применения баз данных, а так же оценить перспективы их развития в настоящее время, то можно получить примерный список наиболее распространенных классов, получивших распространение и применение во всех областях применения баз данных. Этот список будет выглядеть следующим образом: • документографические и документальные применяются во всех базах органов власти и управления • базы данных по промышленной, строительной и сельскохозяйственной продукции • базы данных по экономической и конъюнктурной информации статистическая, кредитно-финансовая, внешнеторговая • фактографические базы социальных данных, включающие сведения о населении и о социальной среде • базы данных транспортных систем • справочные данные для населения и учреждений энциклопедии и справочники, расписания самолетов и поездов, адреса и телефоны граждан и организаций • ресурсные базы данных, включающие фактографическую информацию о природных ресурсах земля, вода, недра, биоресурсы, гидрометеорология, вторичные ресурсы и отходы, экологическая обстановка • фактографические базы и банки научных данных, обеспечивающие фундаментальные научные исследования • фактографические базы данных в области культуры и искусства • лингвистические базы данных, то есть машинные словари разного типа и назначения. |

## 2. Базы данных и файловые системы. Области применения файлов. Потребности информационных систем.

Начнем с того, что с самого начала развития вычислительной техники образовались два основных направления ее использования. Первое направление - применение вычислительной техники для выполнения численных расчетов, которые слишком долго или вообще невозможно производить вручную. Становление этого направления способствовало интенсификации методов численного решения сложных математических задач, развитию класса языков программирования, ориентированных на удобную запись численных алгоритмов, становлению обратной связи с разработчиками новых архитектур ЭВМ.

Второе направление, которое непосредственно касается темы нашего курса, это использование средств вычислительной техники в автоматических или автоматизированных информационных системах. В самом широком смысле информационная система представляет собой программный комплекс, функции которого состоят в поддержке надежного хранения информации в памяти компьютера, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса. Обычно объемы информации, с которыми приходится иметь дело таким системам, достаточно велики, а сама информация имеет достаточно сложную структуру. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т.д.

На самом деле, второе направление возникло несколько позже первого. Это связано с тем, что на заре вычислительной техники компьютеры обладали ограниченными возможностями в части памяти. Понятно, что можно говорить о надежном и долговременном хранении информации только при наличии запоминающих устройств, сохраняющих информацию после выключения электрического питания. Оперативная память этим свойством обычно не обладает. В начале использовались два вида устройств внешней памяти: магнитные ленты и барабаны. При этом емкость магнитных лент была достаточно велика, но по своей физической природе они обеспечивали последовательный доступ к данным. Магнитные же барабаны (они больше всего похожи на современные магнитные диски с фиксированными головками) давали возможность произвольного доступа к данными, но были ограниченного размера.

Легко видеть, что указанные ограничения не очень существенны для чисто численных расчетов. Даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти, чтобы программа работала как можно быстрее.

С другой стороны, для информационных систем, в которых потребность в текущих данных определяется пользователем, наличие только магнитных лент и барабанов неудовлетворительно. Представьте себе покупателя билета, который стоя у кассы должен дождаться полной перемотки магнитной ленты. Одним из естественных требований к таким системам является средняя быстрота выполнения операций.

Как кажется, именно требования к вычислительной технике со стороны нечисленных приложений вызвали появление съемных магнитных дисков с подвижными головками, что явилось революцией в истории вычислительной техники. Эти устройства внешней памяти обладали существенно большей емкостью, чем магнитные барабаны, обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь практически неограниченный архив данных.

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

После этого краткого экскурса в историю файловых систем рассмотрим возможные области их применения. Прежде всего, конечно, файлы применяются для хранения текстовых данных: документов, текстов программ и т.д. Такие файлы обычно образуются и модифицируются с помощью различных текстовых редакторов. Структура текстовых файлов обычно очень проста: это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки).

Файлы с текстами программ используются как входные тексты компиляторов, которые в свою очередь формируют файлы, содержащие объектные модули. С точки зрения файловой системы, объектные файлы также обладают очень простой структурой - последовательность записей или байтов. Система программирования накладывает на эту структуру более сложную и специфичную для этой системы структуру объектного модуля. Подчеркнем, что логическая структура объектного модуля неизвестна файловой системе, эта структура поддерживается программами системы программирования.

Аналогично обстоит дело с файлами, формируемыми редакторами связей и содержащими образы выполняемых программ. Логическая структура таких файлов остается известной только редактору связей и загрузчику - программе операционной системы. Примерно такая же ситуация с файлами, содержащими графическую и звуковую информацию.

Одним словом, файловые системы обычно обеспечивают хранение слабо структурированной информации, оставляя дальнейшую структуризацию прикладным программам. В перечисленных выше случаях использования файлов 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 естественно соответствуют специфике данной прикладной области.

Однако ситуация коренным образом отличается для упоминавшихся в начале лекции информационных систем. Эти системы главным образом ориентированы на хранение, выбор и модификацию постоянно существующей информации. Структура информации зачастую очень сложна, и хотя структуры данных различны в разных информационных системах, между ними часто бывает много общего. На начальном этапе использования вычислительной техники для управления информацией проблемы структуризации данных решались индивидуально в каждой информационной системе. Производились необходимые надстройки над файловыми системами (библиотеки программ), подобно тому, как это делается в компиляторах, редакторах и т.д.

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

Покажем это на примере. Предположим, что мы хотим реализовать простую информационную систему, поддерживающую учет сотрудников некоторой организации. Система должна выполнять следующие действия: выдавать списки сотрудников по отделам, поддерживать возможность перевода сотрудника из одного отдела в другой, приема на работу новых сотрудников и увольнения работающих. Для каждого отдела должна поддерживаться возможность получения имени руководителя этого отдела, общей численности отдела, общей суммы выплаченной в последний раз зарплаты и т.д. Для каждого сотрудника должна поддерживаться возможность выдачи номера удостоверения по полному имени сотрудника, выдачи полного имени по номеру удостоверения, получения информации о текущем соответствии занимаемой должности сотрудника и о размере его зарплаты.

Предположим, что мы решили основывать эту информационную систему на файловой системе и пользоваться при этом одним файлом, расширив базовые возможности файловой системы за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является сотрудник, естественно потребовать, чтобы в этом файле содержалась одна запись для каждого сотрудника. Какие поля должна содержать такая запись? Полное имя сотрудника (СОТР\_ИМЯ), номер его удостоверения (СОТР\_НОМЕР), информацию о его соответствии занимаемой должности (для простоты, "да" или "нет") (СОТР\_СТАТ), размер зарплаты (СОТР\_ЗАРП), номер отдела (СОТР\_ОТД\_НОМЕР). Поскольку мы хотим ограничиться одним файлом, та же запись должна содержать имя руководителя отдела (СОТР\_ОТД\_РУК).

Функции нашей информационной системы требуют, чтобы обеспечивалась возможность многоключевого доступа к этому файлу по уникальным ключам (недублируемым в разных записях) СОТР\_ИМЯ и СОТР\_НОМЕР. Кроме того, должна обеспечиваться возможность выбора всех записей с общем значением СОТР\_ОТД\_НОМЕР, то есть доступ по неуникальному ключу. Для того, чтобы получить численность отдела или общий размер зарплаты, каждый раз при выполнении такой функции информационная система должна будет выбрать все записи о сотрудниках отдела и посчитать соответствующие общие значения.

Таким образом мы видим, что даже для такой простой системы ее реализация на базе файловой системы, во-первых, требует создания достаточно сложной надстройки для многоключевого доступа к файлам, и, во-вторых, вызывает требование существенной избыточности хранения (для каждого сотрудника одного отдела повторяется имя руководителя) и выполнение массовой выборки и вычислений для получения суммарной информации об отделах. Кроме того, если в ходе эксплуатации системы нам захочется, например, выдавать списки сотрудников, получающих заданную зарплату, то придется либо полностью просматривать файл, либо реструктуризовывать его, объявляя ключевым поле СОТР\_ЗАРП.

Первое, что приходит на ум, - это поддерживать два многоключевых файла: СОТРУДНИКИ и ОТДЕЛЫ. Первый файл должен содержать поля СОТР\_ИМЯ, СОТР\_НОМЕР, СОТР\_СТАТ, СОТР\_ЗАРП и СОТР\_ОТД\_НОМЕР, а второй - ОТД\_НОМЕР, ОТД\_РУК, ОТД\_СОТР\_ЗАРП (общий размер зарплаты) и ОТД\_РАЗМЕР (общее число сотрудников в отделе). Большинство неудобств, перечисленных в предыдущем абзаце, будут преодолены. Каждый из файлов будет содержать только недублируемую информацию, необходимости в динамических вычислениях суммарной информации не возникает. Но заметим, что при таком переходе наша информационная система должна обладать некоторыми новыми особенностями, сближающими ее с СУБД.

Прежде всего, система должна теперь знать, что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), должна знать структуру и смысл каждого поля (например, что СОТР\_ОТД\_НОМЕР в файле СОТРУДНИКИ и ОТД\_НОМЕР в файле ОТДЕЛЫ означают одно и то же), а также понимать, что в ряде случаев изменение информации в одном файле должно автоматически вызывать модификацию во втором файле, чтобы их общее содержимое было согласованным. Например, если на работу принимается новый сотрудник, то необходимо добавить запись в файл СОТРУДНИКИ, а также соответствующим образом изменить поля ОТД\_ЗАРП и ОТД\_РАЗМЕР в записи файла ОТДЕЛЫ, описывающей отдел этого сотрудника.

Понятие согласованности данных является ключевым понятием баз данных. Фактически, если информационная система (даже такая простая, как в нашем примере) поддерживает согласованное хранение информации в нескольких файлах, можно говорить о том, что она поддерживает базу данных. Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных. Уже только требование поддержания согласованности данных в нескольких файлах не позволяет обойтись библиотекой функций: такая система должна иметь некоторые собственные данные (метаданные) и даже знания, определяющие целостность данных.

Но это еще не все, что обычно требуют от СУБД. Во-первых, даже в нашем примере неудобно реализовывать такие запросы как "выдать общую численность отдела, в котором работает Петр Иванович Сидоров". Было бы гораздо проще, если бы СУБД позволяла сформулировать такой запрос на близком пользователям языке. Такие языки называются языками запросов к базам данных. Например, на языке SQL наш запрос можно было бы выразить в форме:

SELECT ОТД\_РАЗМЕР

FROM СОТРУДНИКИ, ОТДЕЛЫ

WHERE СОТР\_ИМЯ = "ПЕТР ИВАНОВИЧ СИДОРОВ"

AND СОТР\_ОТД\_НОМЕР = ОТД\_НОМЕР

Таким образом, при формулировании запроса СУБД позволит не задумываться о том, как будет выполняться этот запрос. Среди ее метаданных будет содержаться информация о том, что поле СОТР\_ИМЯ является ключевым для файла СОТРУДНИКИ, а ОТД\_НОМЕР - для файла ОТДЕЛЫ, и система сама воспользуется этим. Если же возникнет потребность в получении списка сотрудников, не соответствующих занимаемой должности, то достаточно предъявить системе запрос

SELECT СОТР\_ИМЯ, СОТР\_НОМЕР

FROM СОТРУДНИКИ

WHERE СОТР\_СТАТ = "НЕТ",

и система сама выполнит необходимый полный просмотр файла СОТРУДНИКИ, поскольку поле СОТР\_СТАТ не является ключевым.

Далее, представьте себе, что в нашей первоначальной реализации информационной системы, основанной на использовании библиотек расширенных методов доступа к файлам, обрабатывается операция регистрации нового сотрудника. Следуя требованиям согласованного изменения файлов, информационная система вставила новую запись в файл СОТРУДНИКИ и собиралась модифицировать запись файла ОТДЕЛЫ, но именно в этот момент произошло аварийное выключение питания. Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии. Потребуется выяснить это (а для этого нужно явно проверить соответствие информации с файлах СОТРУДНИКИ и ОТДЕЛЫ) и привести информацию в согласованное состояние. Настоящие СУБД берут такую работу на себя. Прикладная система не обязана заботиться о корректности состояния базы данных.

Наконец, представим себе, что мы хотим обеспечить параллельную (например, многотерминальную) работу с базой данных сотрудников. Если опираться только на использование файлов, то для обеспечения корректности на все время модификации любого из двух файлов доступ других пользователей к этому файлу будет блокирован (вспомните возможности файловых систем для синхронизации параллельного доступа). Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о сотруднике Иване Сидоровиче Петрове, даже если они будут работать в разных отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

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.

## 3. Основные функции СУБД. Типовая организация современной СУБД.

Современная СУБД выполняет ряд функций, среди которых основными являются следующие:

1. Администрирование базы данных.

СУБД имеют развитые средства администрирования базы данных (определение доступа к базе, ее архивация). В связи с тем, что базы данных проникают сегодня во многие сферы деятельности человека, появилась новая профессия – администратор базы данных, человек, отвечающий за проектирование, создание, использование и сопровождение базы данных. В процессе эксплуатации БД администратор обычно следит за ее функционированием, обеспечивает защиту от несанкционированного доступа к хранимым данным, вносит изменения в структуру базы, контролирует достоверность информации в ней.

2. Непосредственное управление данными во внешней памяти.

Эта функция предоставляет пользователю возможность выполнения основных операций с данными – хранение, извлечение и обновление информации. Она включает в себя обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным. СУБД поддерживает собственную систему именования объектов БД.

3. Управление буферами оперативной памяти.

СУБД обычно работают с БД значительного размера; по крайней мере этот размер обычно существенно больше доступного объема оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. Однако этого недостаточно для целей СУБД. Поэтому в развитых СУБД поддерживается собственный набор буферов оперативной памяти.

4. Управление транзакциями.

Транзакция – это последовательность операций над БД, которые рассматриваются СУБД как единое целое и позволяют добавлять, удалять или обновлять сведения о некотором объекте в базе (по существу это некоторый программный код, написанный на одном из языков управления данными). Либо транзакция успешно выполняется, и СУБД фиксирует изменения БД, произведенные этой транзакцией, либо ни одно из этих изменений никак не отражается на состоянии БД. Например, если в результате транзакции произошел сбой компьютера, база данных попадает в противоречивое положение – некоторые изменения уже внесены, остальные нет.

Транзакция позволяет вернуть базу в первоначальное непротиворечивое состояние (отменить все выполненные изменения).

5. Журнализация.

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее состояние БД после любого аппаратного или программного сбоя (аварийное выключение питания, аварийное завершение работы СУБД или аварийное завершение пользовательской программы). Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Наиболее распространенным методом поддержания надежности хранения является ведение журнала изменений БД.

Журнал – это особая часть БД, недоступная пользователям и поддерживаемая с особой тщательностью (иногда поддерживаются две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. Изменения БД журнализуются следующим образом: запись в журнале соответствует некоторой операции изменения БД (например, операции удаления строки из таблицы реляционной БД). С помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

6. Поддержка языков БД.

СУБД включает язык определения данных, с помощью которого можно определить структуру базы, тип данных в ней, указать ограничения целостности (это язык, с помощью которого задаются различные имена, свойства объектов). Кроме того, СУБД позволяет вставлять, удалять, обновлять и извлекать информацию из базы данных посредством языка управления данными – языка запросов, который позволяет выполнять различные действия с данными, осуществлять их поиск и выборку. Он содержит набор различных операторов (заносить данные, удалять, модифицировать, выбирать и т.д.). Процесс извлечения данных и их обработка скрыты от пользователя.

Стандартным языком наиболее распространенных в настоящее время СУБД является язык SQL (Structured Query Language). Он имеет сразу два компонента: язык определения данных и язык управления данными. Кроме того, одним из языков управления данными является язык QBE – язык запросов по образцу. Подробно о реализаций функций СУБД с помощью языка SQL будет рассказано на отдельных лекциях, посвященных языку SQL.

Естественно, организация типичной СУБД и состав ее компонентов соответствует рассмотренному нами набору функций. Напомним, что мы выделили следующие основные функции СУБД:

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

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

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

Основной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Основной проблемой реляционных СУБД является то, что языки этих систем (а это, как правило, SQL) являются непроцедурными, т.е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия (вспомните примеры из первой лекции). Поэтому компилятор должен решить, каким образом выполнять оператор языка прежде, чем произвести программу. Применяются достаточно сложные методы оптимизации операторов, которые мы подробно рассмотрим в следующих лекциях. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах, но более часто в выполняемом внутреннем машинно-независимом коде. В последнем случае реальное выполнение оператора производится с привлечением подсистемы поддержки времени выполнения, представляющей собой, по сути дела, интерпретатор этого внутреннего языка.

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

## 4. Ранние подходы к организации БД.

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

Заметим, что в этой лекции мы ограничиваемся рассмотрением только общих подходов к организации трех типов ранних систем, а именно, систем, основанных на инвертированных списках, иерархических и сетевых систем управления базами данных. Мы не будем касаться особенностей каких-либо конкретных систем; это привело бы к изложению многих технических деталей, которые, хотя и интересны, находятся несколько в стороне от основной цели нашего курса. Детали можно найти в рекомендованной литературе.

Начнем с некоторых наиболее общих характеристик ранних систем:

* Эти системы активно использовались в течение многих лет, дольше, чем используется какая-либо из реляционных СУБД. На самом деле некоторые из ранних систем используются даже в наше время, накоплены громадные базы данных, и одной из актуальных проблем информационных систем является использование этих систем совместно с современными системами.
* Все ранние системы не основывались на каких-либо абстрактных моделях. Как мы упоминали, 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 Кобол соотносится с уровнем языка 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 пользователя самого 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.

## 5. Реляционный подход к организации БД. Базовые понятия реляционных баз данных.

Когда в предыдущих разделах мы говорили об основных понятиях реляционных баз данных, мы не опирались на какую-либо конкретную реализацию. Эти рассуждения в равной степени относились к любой системе, при построении которой использовался реляционный подход.

Другими словами, мы использовали понятия так называемой реляционной модели данных. Модель данных описывает некоторый набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Хотя понятие модели данных является общим, и можно говорить о иерархической, сетевой, некоторой семантической и т.д. моделях данных, нужно отметить, что это понятие было введено в обиход применительно к реляционным системам и наиболее эффективно используется именно в этом контексте. Попытки прямолинейного применения аналогичных моделей к дореляционным организациям показывают, что реляционная модель слишком "велика" для них, а для постреляционных организаций она оказывается "мала".

Общая характеристика:

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

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

В манипуляционной части модели утверждаются два фундаментальных механизма манипулирования реляционными БД - реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями), а второй - на классическом логическом аппарате исчисления предикатов первого порядка. Мы рассмотрим эти механизмы более подробно на следующей лекции, а пока лишь заметим, что 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.

Понятие **тип данных** в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"), а также специальных "темпоральных" данных (дата, время, временной интервал). Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres). В нашем примере мы имеем дело с данными трех типов: строки символов, целые числа и "деньги".

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

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

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

**Схема отношения** - это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}. Степень или "арность" схемы отношения - мощность этого множества. Степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута).

Схема БД (в структурном смысле) - это набор именованных схем отношений.

**Кортеж**, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

**Отношение** - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой.

Однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть эволюцией схемы базы данных.

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

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

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

## 6. Фундаментальные свойства отношений. Реляционная модель данных.

**Отсутствие кортежей-дубликатов**

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

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

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

**Отсутствие упорядоченности кортежей**

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

**Отсутствие упорядоченности атрибутов**

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

**Атомарность значений атрибутов**

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

В основе реляционных систем лежит реляционная модель данных. Принципы реляционной модели были заложены в 1969–1970 гг. американским ученым Е. Ф. Коддом (E. F. Codd), в то время работавшим в корпорации IBM. Будучи математиком по образованию, он привнес в область управления базами данных строгие математические принципы и точность, которых не хватало ранним системам. Хотя реляционный подход утвердился не сразу, можно отметить, что почти все созданные с конца 70-х гг. продукты баз данных основаны именно на реляционном подходе. Подавляющее большинство научных исследований в области баз данных в течение последних 35 лет также проводилось именно в этом направлении.

Рассматривая и постепенно уточняя основные понятия реляционной модели, будем иметь в виду три компоненты модели данных:

* структуры данных,
* операции, которые можно выполнять над данными, и
* ограничения, связанные с обеспечением целостности данных.

Основной структурой данных в реляционной модели являются таблицы, называемые в реляционной теории отношениями. Собственно от термина отношение (по-английски relation) и произошло само название модели – реляционная.

## 7. Базисные средства манипулирования реляционными данными. Реляционная алгебра. Реляционное исчисление.

Как мы отмечали в предыдущей лекции, в манипуляционной составляющей реляционной модели данных определяются два базовых механизма манипулирования реляционными данными – основанная на теории множеств реляционная алгебра и базирующееся на математической логике (точнее, на исчислении предикатов первого порядка) реляционное исчисление. В свою очередь, обычно выделяются два вида реляционного исчисления – исчисление кортежей и исчисление доменов.

Все эти механизмы обладают одним важным свойством: они замкнуты относительно понятия отношения. Это означает, что выражения реляционной алгебры и формулы реляционного исчисления определяются над отношениями реляционных БД и результатом их "вычисления" также являются отношения (конечно, здесь имеются в виду значения-отношения). В результате любое выражение или формула могут интерпретироваться как отношения, что позволяет использовать их в других выражениях или формулах.

Как мы увидим, алгебра и исчисление обладают большой выразительной мощностью: очень сложные запросы к базе данных могут быть выражены с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления. Именно по этой причине такие механизмы включены в реляционную модель данных. Конкретный язык манипулирования реляционными БД называется реляционно-полным, если любой запрос, формулируемый с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления, может быть сформулирован с помощью одного оператора этого языка.

Известно (и мы не будем это доказывать), что механизмы реляционной алгебры и реляционного исчисления эквивалентны, т. е. для любого допустимого выражения реляционной алгебры можно построить эквивалентную (т. е. производящую такой же результат) формулу реляционного исчисления и наоборот. Почему же в реляционной модели данных присутствуют оба эти механизма?

Дело в том, что они различаются уровнем процедурности. Выражения реляционной алгебры строятся на основе алгебраических операций (высокого уровня), и подобно тому, как интерпретируются арифметические и логические выражения, выражение реляционной алгебры также имеет процедурную интерпретацию. Другими словами, запрос, представленный на языке реляционной алгебры, может быть вычислен на основе выполнения элементарных алгебраических операций с учетом их приоритетности и возможного наличия скобок. Для формулы реляционного исчисления однозначная вычислительная интерпретация, вообще говоря, отсутствует. Формула только ставит условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются в большей степени непроцедурными, или декларативными.

Поскольку механизмы реляционной алгебры и реляционного исчисления эквивалентны, в конкретной ситуации для проверки степени реляционности некоторого языка БД можно пользоваться любым из этих механизмов.

Заметим, что крайне редко алгебра или исчисление принимается в качестве полной основы какого-либо языка БД. Обычно (например, в случае языка SQL) язык основывается на некоторой смеси алгебраических и логических конструкций. Тем не менее знание алгебраических и логических основ языков баз данных часто применяется на практике.

Для экономии времени и места мы не будем вводить какие-либо строгие синтаксические конструкции, а в основном ограничимся рассмотрением материала на содержательном уровне.

Обзор реляционной алгебры 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.

Существует много подходов к определению реляционной алгебры, которые различаются наборами операций и способами их интерпретации, но, в принципе, являются более или менее равносильными. В данном разделе мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом (будем называть ее " алгеброй Кодда "). 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.

Общая интерпретация реляционных операций

Если не вдаваться в некоторые тонкости, которые мы рассмотрим в следующих разделах, то почти для всех операций предложенного выше набора имеется очевидная и простая интерпретация.

При выполнении операции объединения ( UNION ) двух отношений с одинаковыми заголовками производится отношение, включающее все кортежи, которые входят хотя бы в одно из отношений-операндов.

Операция пересечения ( INTERSECT ) двух отношений с одинаковыми заголовками производит отношение, включающее все кортежи, которые входят в оба отношения-операнда.

Отношение, являющееся разностью ( MINUS ) двух отношений с одинаковыми заголовками, включает все кортежи, входящие в отношение-первый операнд, такие, что ни один из них не входит в отношение, которое является вторым операндом.

При выполнении декартова произведения ( TIMES ) двух отношений, пересечение заголовков которых пусто, производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.

Результатом ограничения ( WHERE ) отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.

При выполнении проекции ( PROJECT ) отношения на заданное подмножество множества его атрибутов производится отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда.

При соединении ( JOIN ) двух отношений по некоторому условию образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.

У операции реляционного деления ( DIVIDE BY ) два операнда – бинарное и унарное отношения. Результирующее отношение состоит из унарных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) включает множество значений второго операнда.

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

Операция присваивания ( := ) позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

Поскольку результатом любой реляционной операции (кроме операции присваивания, 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 = DIVIDE BY >= UNION = MINUS

## 8. Проектирование реляционных БД. Нормализация.

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

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером набора ограничений является ограничение первой нормальной формы - значения всех атрибутов отношения атомарны. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

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

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

Основные свойства нормальных форм:

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

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

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

**Определение 1.** *Функциональная зависимость*

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

**Определение 2.** *Полная функциональная зависимость*

Функциональная зависимость R.X (r) R.Y называется полной, если атрибут Y не зависит функционально от любого точного подмножества X.

**Определение 3.***Транзитивная функциональная зависимость*

Функциональная зависимость R.X -> R.Y называется транзитивной, если существует такой атрибут Z, что имеются функциональные зависимости R.X -> R.Z и R.Z -> R.Y и отсутствует функциональная зависимость R.Z --> R.X. (При отсутствии последнего требования мы имели бы "неинтересные" транзитивные зависимости в любом отношении, обладающем несколькими ключами.)

**Определение 4.** *Неключевой атрибут*

Неключевым атрибутом называется любой атрибут отношения, не входящий в состав первичного ключа (в частности, первичного).

**Определение 5.** *Взаимно независимые атрибуты*

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

##### 6.1.1. Вторая нормальная форма

Рассмотрим следующий пример схемы отношения:

СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ

(СОТР\_НОМЕР, СОТР\_ЗАРП, ОТД\_НОМЕР, ПРО\_НОМЕР, СОТР\_ЗАДАН)

Первичный ключ:

СОТР\_НОМЕР, ПРО\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР -> СОТР\_ЗАРП

СОТР\_НОМЕР -> ОТД\_НОМЕР

ОТД\_НОМЕР -> СОТР\_ЗАРП

СОТР\_НОМЕР, ПРО\_НОМЕР -> СОТР\_ЗАДАН

Как видно, хотя первичным ключом является составной атрибут СОТР\_НОМЕР, ПРО\_НОМЕР, атрибуты СОТР\_ЗАРП и ОТД\_НОМЕР функционально зависят от части первичного ключа, атрибута СОТР\_НОМЕР. В результате мы не сможем вставить в отношение СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ кортеж, описывающий сотрудника, который еще не выполняет никакого проекта (первичный ключ не может содержать неопределенное значение). При удалении кортежа мы не только разрушаем связь данного сотрудника с данным проектом, но утрачиваем информацию о том, что он работает в некотором отделе. При переводе сотрудника в другой отдел мы будем вынуждены модифицировать все кортежи, описывающие этого сотрудника, или получим несогласованный результат. Такие неприятные явления называются аномалиями схемы отношения. Они устраняются путем нормализации.

**Определение 6.** *Вторая нормальная форма (в этом определении предполагается, что единственным ключом отношения является первичный ключ)*

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

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

СОТРУДНИКИ-ОТДЕЛЫ (СОТР\_НОМЕР, СОТР\_ЗАРП, ОТД\_НОМЕР)

Первичный ключ:

СОТР\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР -> СОТР\_ЗАРП

СОТР\_НОМЕР -> ОТД\_НОМЕР

ОТД\_НОМЕР -> СОТР\_ЗАРП

СОТРУДНИКИ-ПРОЕКТЫ (СОТР\_НОМЕР, ПРО\_НОМЕР, СОТР\_ЗАДАН)

Первичный ключ:

СОТР\_НОМЕР, ПРО\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР, ПРО\_НОМЕР -> CОТР\_ЗАДАН

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

Если допустить наличие нескольких ключей, то определение 6 примет следующий вид:

**Определение 6~**

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

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

##### 6.1.2. Третья нормальная форма

Рассмотрим еще раз отношение СОТРУДНИКИ-ОТДЕЛЫ, находящееся в 2NF. Заметим, что функциональная зависимость СОТР\_НОМЕР -> СОТР\_ЗАРП является транзитивной; она является следствием функциональных зависимостей СОТР\_НОМЕР -> ОТД\_НОМЕР и ОТД\_НОМЕР -> СОТР\_ЗАРП. Другими словами, заработная плата сотрудника на самом деле является характеристикой не сотрудника, а отдела, в котором он работает (это не очень естественное предположение, но достаточное для примера).

В результате мы не сможем занести в базу данных информацию, характеризующую заработную плату отдела, до тех пор, пока в этом отделе не появится хотя бы один сотрудник (первичный ключ не может содержать неопределенное значение). При удалении кортежа, описывающего последнего сотрудника данного отдела, мы лишимся информации о заработной плате отдела. Чтобы согласованным образом изменить заработную плату отдела, мы будем вынуждены предварительно найти все кортежи, описывающие сотрудников этого отдела. Т.е. в отношении СОТРУДИКИ-ОТДЕЛЫ по-прежнему существуют аномалии. Их можно устранить путем дальнейшей нормализации.

**Определение 7.** *Третья нормальная форма. (Снова определение дается в предположении существования единственного ключа.)*

Отношение R находится в третьей нормальной форме (3NF) в том и только в том случае, если находится в 2NF и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

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

СОТРУДНИКИ (СОТР\_НОМЕР, ОТД\_НОМЕР)

Первичный ключ:

СОТР\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР -> ОТД\_НОМЕР

ОТДЕЛЫ (ОТД\_НОМЕР, СОТР\_ЗАРП)

Первичный ключ:

ОТД\_НОМЕР

Функциональные зависимости:

ОТД\_НОМЕР -> СОТР\_ЗАРП

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

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

**Определение 7~**

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

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

##### 6.1.3. Нормальная форма Бойса-Кодда

Рассмотрим следующий пример схемы отношения:

СОТРУДНИКИ-ПРОЕКТЫ (СОТР\_НОМЕР, СОТР\_ИМЯ, ПРО\_НОМЕР, СОТР\_ЗАДАН)

Возможные ключи:

СОТР\_НОМЕР, ПРО\_НОМЕР

СОТР\_ИМЯ, ПРО\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР -> CОТР\_ИМЯ

СОТР\_НОМЕР -> ПРО\_НОМЕР

СОТР\_ИМЯ -> CОТР\_НОМЕР

СОТР\_ИМЯ -> ПРО\_НОМЕР

СОТР\_НОМЕР, ПРО\_НОМЕР -> CОТР\_ЗАДАН

СОТР\_ИМЯ, ПРО\_НОМЕР -> CОТР\_ЗАДАН

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

В соответствии с определением 7~ отношение СОТРУДНИКИ-ПРОЕКТЫ находится в 3NF. Однако тот факт, что имеются функциональные зависимости атрибутов отношения от атрибута, являющегося частью первичного ключа, приводит к аномалиям. Например, для того, чтобы изменить имя сотрудника с данным номером согласованным образом, нам потребуется модифицировать все кортежи, включающие его номер.

**Определение 8.** *Детерминант*

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

**Определение 9.** *Нормальная форма Бойса-Кодда*

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

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

СОТРУДНИКИ (СОТР\_НОМЕР, СОТР\_ИМЯ)

Возможные ключи:

СОТР\_НОМЕР

СОТР\_ИМЯ

Функциональные зависимости:

СОТР\_НОМЕР -> CОТР\_ИМЯ

СОТР\_ИМЯ -> СОТР\_НОМЕР

СОТРУДНИКИ-ПРОЕКТЫ (СОТР\_НОМЕР, ПРО\_НОМЕР, СОТР\_ЗАДАН)

Возможный ключ:

СОТР\_НОМЕР, ПРО\_НОМЕР

Функциональные зависимости:

СОТР\_НОМЕР, ПРО\_НОМЕР -> CОТР\_ЗАДАН

Возможна альтернативная декомпозиция, если выбрать за основу СОТР\_ИМЯ. В обоих случаях получаемые отношения СОТРУДНИКИ и СОТРУДНИКИ-ПРОЕКТЫ находятся в BCNF, и им не свойственны отмеченные аномалии.

## 9. Основные понятия модели «Сущность-Связь». ER-диаграммы.

Моделирование структуры базы данных при помощи алгоритма нормализации, описанного в предыдущих главах, имеет серьезные недостатки:

1. Первоначальное размещение всех атрибутов в одном отношении является очень неестественной операцией. Интуитивно разработчик сразу проектирует несколько отношений в соответствии с обнаруженными сущностями. Даже если совершить насилие над собой и создать одно или несколько отношений, включив в них все предполагаемые атрибуты, то совершенно неясен смысл полученного отношения.
2. Невозможно сразу определить полный список атрибутов. Пользователи имеют привычку называть разными именами одни и те же вещи или наоборот, называть одними именами разные вещи.
3. Для проведения процедуры нормализации необходимо выделить зависимости атрибутов, что тоже очень нелегко, т.к. необходимо *явно выписать все зависимости*, даже те, которые являются очевидными.

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

Первый вариант модели сущность-связь был предложен в 1976 г. Питером Пин-Шэн Ченом [37]. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. По сути, все варианты диаграмм сущность-связь исходят из одной идеи - рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Мы опишем работу с ER-диаграммами близко к нотации Баркера, как довольно легкой в понимании основных идей. Данная глава является скорее иллюстрацией методов семантического моделирования, чем полноценным введением в эту область.

#### Основные понятия ER-диаграмм

*Определение 1*. ***Сущность*** - это класс однотипных объектов, информация о которых должна быть учтена в модели.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе.

Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

Каждая сущность в модели изображается в виде прямоугольника с наименованием:



**Рис. 1**

*Определение 2*. ***Экземпляр сущности*** - это конкретный представитель данной сущности.

Например, представителем сущности "Сотрудник" может быть "Сотрудник Иванов".

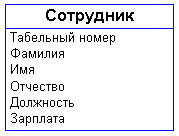
Экземпляры сущностей должны быть *различимы*, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

*Определение 3*. ***Атрибут сущности*** - это именованная характеристика, являющаяся некоторым свойством сущности.

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

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

Атрибуты изображаются в пределах прямоугольника, определяющего сущность:

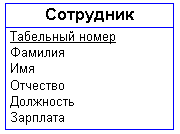


**Рис. 2**

*Определение 4*. ***Ключ сущности*** - это *неизбыточный* набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности. Неизбыточность заключается в том, что удаление любого атрибута из ключа нарушается его уникальность.

Сущность может иметь несколько различных ключей.

Ключевые атрибуты изображаются на диаграмме подчеркиванием:



**Рис. 3**

*Определение 5*. ***Связь*** - это некоторая ассоциация между *двумя* сущностями. Одна сущность может быть связана с другой сущностью или сама с собою.

Связи позволяют по одной сущности находить другие сущности, связанные с нею.

Например, связи между сущностями могут выражаться следующими фразами - "СОТРУДНИК может иметь несколько ДЕТЕЙ", "каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ".

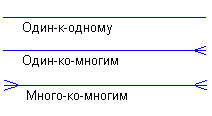
Графически связь изображается линией, соединяющей две сущности:



**Рис. 4**

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

Каждая связь может иметь один из следующих ***типов связи***:



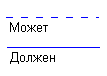
**Рис. 5**

Связь типа ***один-к-одному*** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа ***один-ко-многим*** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется ***родительской***, правая (со стороны "много") - ***дочерней***. Характерный пример такой связи приведен на Рис. 4.

Связь типа ***много-ко-многим*** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является *временным* типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

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



**Рис. 6**

Модальность "***может***" означает, что экземпляр одной сущности *может быть связан* с одним или несколькими экземплярами другой сущности, *а может быть и не связан* ни с одним экземпляром.

Модальность "***должен***" означает, что экземпляр одной сущности *обязан быть связан не менее чем с одним* экземпляром другой сущности.

Связь может иметь *разную модальность* с разных концов (как на Рис. 4).

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

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

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на Рис. 4 читается так:

Слева направо: "каждый сотрудник может иметь несколько детей".

Справа налево: "Каждый ребенок обязан принадлежать ровно одному сотруднику".

#### Пример разработки простой ER-модели

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

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

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

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

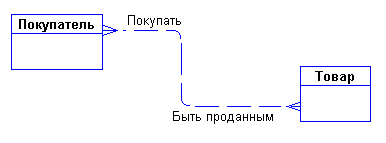
Например, в ходе беседы с менеджером по продажам, выяснилось, что он (менеджер) считает, что проектируемая система должна выполнять следующие действия:

* Хранить информацию о покупателях.
* Печатать накладные на отпущенные товары.
* Следить за наличием товаров на складе.

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

* *Покупатель* - явный кандидат на сущность.
* *Накладная* - явный кандидат на сущность.
* *Товар* - явный кандидат на сущность
* *(?)Склад* - а вообще, сколько складов имеет фирма? Если несколько, то это будет кандидатом на новую сущность.
* *(?)Наличие товара* – это, скорее всего, атрибут, но атрибут какой сущности?

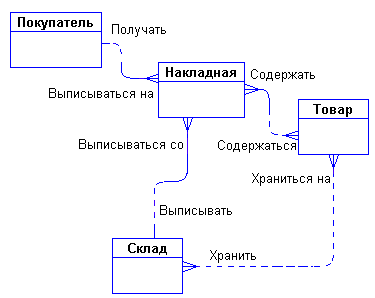
Сразу возникает очевидная связь между сущностями - "покупатели могут покупать много товаров" и "товары могут продаваться многим покупателям". Первый вариант диаграммы выглядит так:



**Рис. 7**

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

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



**Рис. 8**

Пора подумать об атрибутах сущностей. Беседуя с сотрудниками фирмы, мы выяснили следующее:

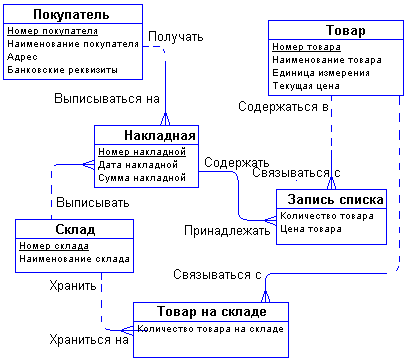
* Каждый покупатель является юридическим лицом и имеет наименование, адрес, банковские реквизиты.
* Каждый товар имеет наименование, цену, а также характеризуется единицами измерения.
* Каждая накладная имеет уникальный номер, дату выписки, список товаров с количествами и ценами, а также общую сумму накладной. Накладная выписывается с определенного склада и на определенного покупателя.
* Каждый склад имеет свое наименование.
* Снова выпишем все существительные, которые будут потенциальными атрибутами, и проанализируем их:
* *Юридическое лицо* - термин риторический, мы не работаем с физическими лицами. Не обращаем внимания.
* *Наименование покупателя* - явная характеристика покупателя.
* *Адрес* - явная характеристика покупателя.
* *Банковские реквизиты* - явная характеристика покупателя.
* *Наименование товара* - явная характеристика товара.
* *(?)Цена товара* - похоже, что это характеристика товара. Отличается ли эта характеристика от цены в накладной?
* *Единица измерения* - явная характеристика товара.
* *Номер накладной* - явная уникальная характеристика накладной.
* *Дата накладной* - явная характеристика накладной.
* *(?)Список товаров в накладной* - список не может быть атрибутом. Вероятно, нужно выделить этот список в отдельную сущность.
* *(?)Количество товара в накладной* - это явная характеристика, но характеристика чего? Это характеристика не просто "товара", а "товара в накладной".
* *(?)Цена товара в накладной* - опять же это должна быть не просто характеристика товара, а характеристика товара в накладной. Но цена товара уже встречалась выше - это одно и то же?
* *Сумма накладной* - явная характеристика накладной. Эта характеристика не является независимой. Сумма накладной равна сумме стоимостей всех товаров, входящих в накладную.
* *Наименование склада* - явная характеристика склада.

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

С возникающим понятием "Список товаров в накладной" все довольно ясно. Сущности "Накладная" и "Товар" связаны друг с другом отношением типа много-ко-многим. Такая связь, как мы отмечали ранее, должна быть расщеплена на две связи типа один-ко-многим. Для этого требуется дополнительная сущность. Этой сущностью и будет сущность "Список товаров в накладной". Связь ее с сущностями "Накладная" и "Товар" характеризуется следующими фразами - "каждая накладная обязана иметь несколько записей из списка товаров в накладной", "каждая запись из списка товаров в накладной обязана включаться ровно в одну накладную", "каждый товар может включаться в несколько записей из списка товаров в накладной", " каждая запись из списка товаров в накладной обязана быть связана ровно с одним товаром". Атрибуты "Количество товара в накладной" и "Цена товара в накладной" являются атрибутами сущности " Список товаров в накладной".

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

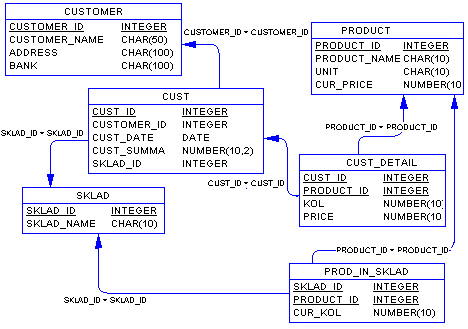
Теперь можно внести все это в диаграмму:



**Рис. 9**

#### Концептуальные и физические ER-модели

Разработанный выше пример ER-диаграммы является примером ***концептуальной диаграммы***. Это означает, что диаграмма *не учитывает* особенности конкретной СУБД. По данной концептуальной диаграмме можно построить ***физическую диаграмму***, которая уже будут учитываться такие особенности СУБД, как допустимые типы и наименования полей и таблиц, ограничения целостности и т.п. Физический вариант диаграммы, приведенной на Рис. 9 может выглядеть, например, следующим образом:



**Рис. 10**

На данной диаграмме каждая сущность представляет собой таблицу базы данных, каждый атрибут становится колонкой соответствующей таблицы. Обращаем внимание на то, что во многих таблицах, например, "CUST\_DETAIL" и "PROD\_IN\_SKLAD", соответствующих сущностям "Запись списка накладной" и "Товар на складе", появились новые атрибуты, которых не было в концептуальной модели - это ключевые атрибуты родительских таблиц, ***мигрировавших*** в дочерние таблицы для того, чтобы обеспечить связь между таблицами посредством внешних ключей.

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

## 10. Система System R. Основные цели System R.

Основными целями разработчиков System R являлись следующие:

1. обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
2. обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
3. поддерживать динамически изменяемую среду баз данных, в которой отношения, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
4. обеспечить возможность параллельной работы с одной базой данных многих пользователей с допущением параллельной модификации объектов базы данных при наличии необходимых средств защиты целостности базы данных;
5. обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
6. 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;
7. обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Прежде всего отметим, что в основном поставленные цели при разработке System R были достигнуты. Рассмотрим теперь, какими средствами были достигнуты эти цели, и как более точно можно интерпретировать их в контексте System R.

Основой System R является реляционный язык SQL. Иногда его называют языком запросов или языком манипулирования данными, но на самом деле его возможности гораздо шире. Средствами SQL (с соответствующей системной поддержкой) решаются многие из поставленных целей. Язык SQL включает средства динамической компиляции запросов, на основе чего возможно построение диалоговых систем обработки запросов. Допускается динамическая параметризация статически откомпилированных запросов, в результате чего возможно построение эффективных (не требующих динамической компиляции) диалоговых систем со стандартными наборами (параметризуемых) запросов.

Средствами SQL определяются все доступные пользователю объекты баз данных: таблицы, индексы, представления. Имеются средства уничтожения любого такого объекта. Соответствующие операторы языка могут выполняться в любой момент, и возможность выполнения операции данным пользователем зависит от ранее предоставленных ему прав.

Что касается целостности баз данных, то в System R под целостным состоянием базы данных понимается состояние, удовлетворяющее набору сохраняемых при базе данных предикатов целостности. Эти предикаты, называемые в System R условиями целостности (assertions), задаются также средствами языка SQL. Любое предложение языка выполняется в пределах некоторой транзакции - неделимой в смысле состояния базы данных последовательности предложений языка. Неделимость означает, что все изменения, 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 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 инициированный по причине нарушения условия целостности, производится до ближайшей точки сохранения, в которой условия целостности соблюдены. (Заметим, что средства установки точек сохранения отсутствуют в коммерческих расширениях System R).

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

В языке SQL имеется средство определения так называемых условных воздействий (triggers), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. Условное воздействие - это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения. Особенно существенно наличие такого аппарата в связи с наличием рассматриваемых 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. Заметим, что, за исключением ранних публикаций по System R, реализация механизма условных воздействий нигде не описывалась, хотя в принципе подходы к реализации достаточно понятны. Этот механизм не реализован в коммерческих системах, возникших на базе System R. Видимо, это связано с возникающими дополнительными непредсказуемыми для пользователей накладными расходами при выполнении транзакций.

Язык SQL содержит средства определения представлений. Представление - это запомненный именованный запрос на выборку данных (из одной или нескольких таблиц). Поскольку SQL - это реляционный язык, то результатом выполнения любого запроса на выборку является таблица, и поэтому концептуально можно относиться к любому представлению как к таблице (при определении представления можно, в частности, присвоить имена полям этой таблицы). В языке допускается использование ранее определенных представлений практически везде, где допускается использование таблиц (с некоторыми ограничениями по поводу возможностей модификации через представления). Наличие возможности определять представления в совокупности с развитой системой авторизации позволяет ограничить доступ некоторых пользователей к базе данных выделенным набором представлений.

Авторизация доступа к базе данных основана также на средствах SQL. При создании любого объекта базы данных выполняющий эту операцию пользователь становится полновластным владельцем этого объекта, т.е. может выполнять по отношению к этому объекту любую функцию из предопределенного набора. Далее этот пользователь может выполнить оператор SQL, означающий передачу всех его прав на этот объект (или их подмножества) любому другому пользователю. В частности, этому пользователю может быть передано право на передачу всех переданных ему прав (или их части) третьему пользователю и т.д. Одним из прав пользователя по отношению к объекту является право на изъятие у других пользователей всех или некоторых прав, которые ранее им были переданы. Эта операция распространяется транзитивно на всех дальнейших наследников этих прав.

Наличие в языке средств определения представлений и авторизации в принципе позволяет обойтись при эксплуатации System R без традиционного администратора баз данных, поскольку практически все системные действия производятся на основе средств SQL. Тем не менее, если организационно администратор баз данных требуется, то его работа достаточно упрощается за счет унифицированного набора средств управления. Кроме того, в System R каталоги баз данных поддерживаются также в виде таблиц, и к ним применены все запросы языка SQL. Заметим, что в коммерческих СУБД появился ряд дополнительных утилит, не связанных с языком SQL (например, утилиты сбора статистики или массовой загрузки базы данных), и в этих системах, видимо, без администратора базы данных не обойтись.

По части обеспечения параллельной работы многих пользователей с одной базой данных, основной подход System R состоит в том, что пользователь не обязан знать о наличии других, конкурирующих с ним за доступ к базе данных, пользователей, 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 захватов по отношению ко всем изменяемым объектам базы данных. Имеется ряд тонкостей, связанных с такой синхронизацией, на которых мы остановимся ниже.

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 мы отмечали, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. На информации журнала основано восстановление базы данных и после сбоев другого рода. Управление журнализацией и восстановлением в System R весьма интересно, применяемые методы в ряде случаев отличаются от методов, используемых в других СУБД.

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со спецификой физической организации баз данных на внешней памяти, буферизацией используемых страниц базы данных в оперативной памяти и развитой техникой оптимизации запросов, сформулированных на SQL, производимой на стадии их компиляции.

Структурная организация System R вполне согласуется с поставленными при ее разработке целями и выбранными решениями. Основными структурными компонентами System R являются система управления реляционной памятью (Relational Storage System - RSS) и компилятор запросов языка SQL. 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 отношений, 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 заменить на другой язык, но он должен обладать примерно такой же семантикой.

Далее мы последовательно рассмотрим особенности организации RSS, процесс компиляции и оптимизации запросов и технику выполнения откомпилированных транзакций (включая отмеченную выше возможность динамической компиляции запросов).

## 11. Структуры внешней памяти, методы организации индексов. Хранение отношений.

Реляционные СУБД обладают рядом особенностей, влияющих на организацию внешней памяти. К наиболее важным особенностям можно отнести следующие:

* Наличие двух уровней системы: уровня непосредственного управления данными во внешней памяти (а также обычно управления буферами оперативной памяти, управления транзакциями и журнализацией изменений БД) и языкового уровня (например, уровня, реализующего язык SQL). При такой организации подсистема нижнего уровня должна поддерживать во внешней памяти набор базовых структур, конкретная интерпретация которых входит в число функций подсистемы верхнего уровня.
* Поддержание отношений-каталогов. Информация, связанная с именованием объектов базы данных и их конкретными свойствами (например, структура ключа индекса), поддерживается подсистемой языкового уровня. С точки зрения структур внешней памяти отношение-каталог ничем не отличается от обычного отношения базы данных.
* Регулярность структур данных. Поскольку основным объектом реляционной модели данных является плоская таблица, главный набор объектов внешней памяти может иметь очень простую регулярную структуру.
* При этом необходимо обеспечить возможность эффективного выполнения операторов языкового уровня как над одним отношением (простые селекция и проекция), так и над несколькими отношениями (наиболее распространено и трудоемко соединение нескольких отношений). Для этого во внешней памяти должны поддерживаться дополнительные "управляющие" структуры - индексы.
* Наконец, для выполнения требования надежного хранения баз данных необходимо поддерживать избыточность хранения данных, что обычно реализуется в виде журнала изменений базы данных.

Соответственно возникают следующие разновидности объектов во внешней памяти базы данных:

* строки отношений - основная часть базы данных, большей частью непосредственно видимая пользователям;
* управляющие структуры - индексы, создаваемые по инициативе пользователя (администратора) или верхнего уровня системы из соображений повышения эффективности выполнения запросов и обычно автоматически поддерживаемые нижним уровнем системы;
* журнальная информация, поддерживаемая для удовлетворения потребности в надежном хранении данных;
* служебная информация, поддерживаемая для удовлетворения внутренних потребностей нижнего уровня системы (например, информация о свободной памяти).

Мы рассматривали на примерах System R и Ingres два альтернативных подхода к организации реляционной СУБД с точки разделения функций между различными компонентами. Напомним, что в СУБД System R существовала интегрированная подсистема управления данными, транзакциями и журнализацией, в то время как в Ingres управление данными, было отделено от управления транзакциями и журнализацией.

У обоих этих подходов имеются свои преимущества и недостатки. Подход System R позволяет использовать более эффективные методы за счет совместного решения проблем физической и логической синхронизации, использовании общих протоколов при управлении буферами и журнализации и т.д. Но при этом в некотором смысле подсистема нижнего уровня становится монолитом; при самой удачной ее структуризации компоненты остаются связанными общими протоколами взаимодействия. Непродуманные локальные изменения одного компонента могут привести к фатальным последствиям для всей системы. Подход Ingres позволяет упростить структуру системы и сделать ее более гибкой, но это возможно только за счет огрубления алгоритмов: применения более грубых методов управления транзакциями; жестких протоколов журнализации и т.д.

В конечном счете любая конкретная система основывается на конкретном комплексном решении. Мы рассматриваем здесь фрагменты таких решений (эскизы).

## 12. Транзакции и целостность баз данных. Сериализация транзакций.

Поддержка механизма транзакций – показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей в многопользовательских системах. Часто эти два аспекта рассматриваются по отдельности, но на самом деле они взаимосвязаны, что и будет показано в этой лекции.

**Общее понятие транзакции и основные характеристики транзакций**

Более точно, в современных СУБД поддерживается понятие транзакции, характеризуемое аббревиатурой 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). Свойство атомарности, которое часто называют свойством “все или ничего”, позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных (в общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения).
* *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. Это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки согласованности и явным образом проверять ограничения целостности. (С точки зрения автора, в контексте баз данных термины *согласованность* и *целостность* эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности, т.е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.)
* *Изоляция (Isolation).* Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции *T1* не должны быть видны никакой другой транзакции *T2* до тех пор, пока транзакция *T1* не завершится успешным образом.
* *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. Этому аспекту транзакционных систем посвящается лекция 14.

Заметим, что хотя с точки зрения обеспечения целостности баз данных механизм транзакций следовало бы поддерживать в персональных СУБД, на практике это обычно не выполняется. Поэтому при переходе от персональных к многопользовательским СУБД пользователи сталкиваются с необходимостью четкого понимания природы транзакций.

#### Атомарность транзакций

В этом смысле под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в состоянии базы данных, либо воздействие всех этих операторов полностью отсутствует.

Лозунгом транзакции является «Все или ничего»: при завершении транзакции оператором COMMIT(высокоуровневый аналог операции END TRANSACTION в интерфейсе RSS, см. лекцию 12) 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, см. лекцию 12) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Каким образом в СУБД поддерживаются индивидуальные откаты транзакций, описывается в лекции 14.

#### Транзакции и целостность баз данных

Понятие транзакции имеет непосредственную связь с понятием целостности базы данных. Очень часто база данных может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения базы данных. Например, в базе данных СЛУЖАЩИЕ-ОТДЕЛЫ (см. лекцию 1) естественным ограничением целостности является совпадение значения атрибута ОТД\_РАЗМЕР в кортеже таблицы ОТДЕЛЫ, описывающей данный отдел (например, отдел 625), с числом кортежей таблицы СЛУЖАЩИЕ, таких, что значение поля СЛУ\_ОТД\_НОМЕР равно 625. Как в этом случае принять на работу в отдел 625 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в таблице СОТРУДНИКИ или модификация существующего кортежа в отношении ОТДЕЛЫ, после выполнения операции база данных окажется в нецелостном состоянии.

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

Более точно, 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). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более чем на 100000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности – это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор ROLLBACK 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).

Заметим, что концептуально в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако в реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных СЛУЖАЩИЕ-ОТДЕЛЫв ней не выполнялись операторы вставки или удаления кортежей из отношения СЛУЖАЩИЕ, то проверять упоминавшееся выше ограничение целостности не требуется (а для проверки подобных ограничений требуется достаточно большая работа).

Понятно, что описанный механизм поддержки целостности баз данных обеспечивает требуемое свойство транзакций: никакая транзакция не может быть зафиксирована, если ее действия нарушили целостность базы данных. Однако в этом подходе имеются два серьезных дефекта.

Во-первых, если при выполнении транзакции не устанавливать точки сохранения и не проверять периодически соответствие текущего состояния базы данных (с точки зрения данной транзакции) ограничениям целостности, то долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора COMMIT. Конечно, это означает непроизводительный расход системных ресурсов и времени пользователей. Во-вторых, чем длиннее транзакция, модифицирующая состояние базы данных, тем потенциально больше ограничений целостности придется проверять при ее завершении и тем накладнее становится оператор COMMIT.

Простое и элегантное решение этой проблемы предлагается в [[1.5]](http://citforum.ru/database/advanced_intro/86.shtml#ref.1.5). Авторы предлагают отказаться от откладываемых ограничений целостности базы данных, а вместо этого ввести составные операторы изменения базы данных (нечто наподобие блоков BEGIN … END, поддерживаемых в языках программирования). После выполнения каждого такого блока (или отдельного оператора изменения базы данных, используемого без операторов начала и конца блока) база данных должна находится в целостном состоянии. Если составной оператор нарушает ограничение целостности, то он целиком отвергается, и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. Понятно, что при использовании такого подхода при выполнении оператора COMMIT не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии.

Интересно, что для реализации описанного подхода не требуются какие-либо новые механизмы, кроме точек сохранения транзакции, насильственной проверки ограничений целостности и частичных откатов транзакций, а отмеченные ранее проблемы снимаются. К сожалению, насколько известно автору данной книги, этот подход на практике пока не применяется.

#### Изолированность транзакций

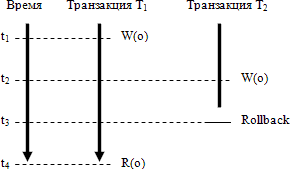
В многопользовательских системах с одной базой данных одновременно может работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с базой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

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

При соблюдении обязательного требования поддержки целостности базы данных возможно наличие нескольких уровней изолированности транзакций. Заметим, что впервые эти уровни изолированности транзакций были установлены и описаны участниками проекта System R.

##### Отсутствие потерянных изменений (первый уровень изолированности)

Рассмотрим сценарий совместного выполнения двух транзакций, показанный на рис. 13.1. В момент времени t1 транзакция T1 изменяет объект базы данных o (выполняет операцию W(o)). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 также изменяет объект o. В момент времени t3 > t2транзакция T2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности).

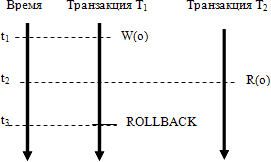
   
*Рис. 13.1.* Потерянные изменения

Тогда при повторном чтении объекта o (выполнении операции R(o)) в момент времени t4 > t3 транзакцияT1 не видит своих изменений этого объекта, произведенных ранее (в частности, из-за этого может не удастся фиксация этой транзакции, что, возможно, повлечет потерю изменений у еще одной транзакции и т.д.).

Такая ситуация называется ситуацией *потерянных изменений*. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции T1 требуется, чтобы до завершения транзакции T1 никакая другая транзакция не могла изменять никакой измененный транзакцией T1 объект o (в частности, достаточно заблокировать доступ по изменению к объекту o до завершения транзакции T1). Отсутствие потерянных изменений является минимальным требованием к СУБД при обеспечении изолированности одновременно выполняемых транзакций.

##### Отсутствие чтения «грязных» данных (второй уровень изолированности)

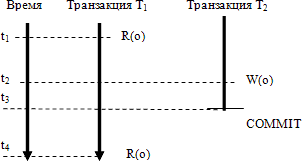
Рассмотрим сценарий совместного выполнения транзакций T1 и T2, показанный на рис. 13.2. В момент времени t1 транзакция T1 изменяет объект базы данных o (выполняет операцию W(o)). В момент времени t2 > t1 транзакция T2 читает объект o (выполняет операцию R(o)). Поскольку транзакция T1 еще не завершена, транзакция T2 видит несогласованные *«грязные» данные*. В частности, в момент времени t3 >t2 транзакция T1 может завершиться откатом (например, по причине нарушения ограничений целостности).

   
*Рис. 13.2.* «Грязные» чтения

Эта ситуация тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и имеет право видеть только согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции T1, изменившей объект базы данных o, никакая другая транзакция не должна читать объект o (например, достаточно заблокировать доступ по чтению к объекту o до завершения изменившей его транзакции T1).

##### Отсутствие неповторяющихся чтений (третий уровень изоляции)

Рассмотрим сценарий совместного выполнения транзакций T1 и T2, показанный на рис. 13.3. В момент времени t1 транзакция T1 читает объект базы данных o (выполняет операцию R(o)). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 изменяет объект o (выполняет операцию W(o)) и успешно завершается оператором COMMIT. В момент времени t3 > t2 транзакция T1 повторно читает объект o и видит его измененное состояние.

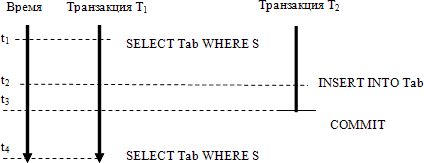
   
*Рис. 13.3.* Неповторяющиеся чтения

Чтобы избежать *неповторяющихся чтений*, до завершения транзакции T1 никакая другая транзакция не должна изменять объект o (для этого достаточно заблокировать доступ по записи к объекту o до завершения транзакции T1). Часто это является максимальным требованием к средствам обеспечения изолированности транзакций, хотя, как будет видно немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (кстати, соответствующие операторы были предусмотрены уже в стандарте SQL:1992). Как уже отмечалось, для корректного соблюдения ограничений целостности достаточен первый уровень. Существует ряд приложений, которым хватает первого уровня изолированности (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

##### Проблема фантомов

К более тонким проблемам изолированности транзакций относится так называемая проблема *кортежей-«фантомов»*, приводящая к ситуациям, которые также противоречат изолированности пользователей. Рассмотрим сценарий, показанный на рис. 13.4.

   
*Рис. 13.4.* Проблема фантомов

В момент времени t1 транзакция T1 выполняет оператор выборки кортежей таблицы Tab с условием выборки S (т.е. выбирается часть кортежей таблицы Tab, удовлетворяющих условию S). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 вставляет в таблицу Tab новый кортеж r, удовлетворяющий условию S, и успешно завершается. В момент времени t3 > t2 транзакция T1 повторно выполняет тот же оператор выборки, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора.

Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

#### Сериализация транзакций

Чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.

Пусть в системе одновременно выполняется некоторое множество транзакций S = {T1, T2, …, T*n*}. План (способ) выполнения набора транзакций S (в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций) называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций (T*i*1, T*i*2, …, T*in*).

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

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

Между транзакциями T1 и T2 могут существовать следующие виды конфликтов:

* W/W – транзакция T2 пытается изменять объект, измененный не закончившейся транзакцией T1(наличие такого конфликта может привести к возникновению ситуации потерянных изменений);
* R/W – транзакция T2 пытается изменять объект, прочитанный не закончившейся транзакцией T1(наличие такого конфликта может привести к возникновению ситуации неповторяющихся чтений);
* W/R – транзакция T2 пытается читать объект, измененный не закончившейся транзакцией T1(наличие такого конфликта может привести к возникновению ситуации «грязного» чтения).

Практические методы сериализации транзакций основываются на учете этих конфликтов.

## 13. Изолированность пользователей.

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

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

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

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

Второй уровень - отсутствие чтения "грязных данных". Рассмотрим следующий сценарий совместного выполнения транзакций 1 и 2. Транзакция 1 изменяет объект базы данных A. Параллельно с этим транзакция 2 читает объект A. Поскольку операция изменения еще не завершена, транзакция 2 видит несогласованные "грязные" данные (в частности, операция транзакции 1 может быть отвернута при проверке немедленно проверяемого ограничения целостности). Это тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и в праве ожидать видеть согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции 1, изменившей объект A, никакая другая транзакция не должна читать объект A (минимальным требованием является блокировка чтения объекта A до завершения операции его изменения в транзакции 1).

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

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

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

## 14. Язык SQL, функции и основные возможности. Стандартизация SQL.

Язык *SQL* является основой многих *СУБД*, т.к. отвечает за физическое структурирование и *запись* данных на *диск*, а также за *чтение данных* с диска, позволяет принимать *SQL*- *запросы* от других компонентов *СУБД* и пользовательских приложений. Таким образом, *SQL*– мощный инструмент, который обеспечивает пользователям, программам и вычислительным системам *доступ* к информации, содержащейся в *реляционных базах данных*.

Основные достоинства языка *SQL* заключаются в следующем:

* стандартность – как уже было сказано, использование языка SQL в программах стандартизировано международными организациями;
* независимость от конкретных *СУБД* – все распространенные *СУБД* используют SQL, т.к. *реляционную базу данных* можно перенести с одной *СУБД* на другую с минимальными доработками;
* возможность переноса с одной вычислительной системы на другую – *СУБД* может быть ориентирована на различные вычислительные системы, однако приложения, созданные с помощью SQL, допускают использование как для *локальных БД*, так и для крупных многопользовательских систем;
* реляционная основа языка – SQL является языком *реляционных БД*, поэтому он стал популярным тогда, когда получила широкое распространение реляционная модель представления данных. Табличная структура *реляционной БД* хорошо понятна, а потому язык SQL прост для изучения;
* возможность создания *интерактивных запросов* – SQL обеспечивает пользователям немедленный доступ к данным, при этом в интерактивном режиме можно получить результат *запроса* за очень короткое время без написания сложной программы;
* возможность программного доступа к БД – язык SQL легко использовать в приложениях, которым необходимо обращаться к *базам данных*. Одни и те же операторы SQL употребляются как для интерактивного, так и программного доступа, поэтому части программ, содержащие обращение к БД, можно вначале проверить в интерактивном режиме, а затем встраивать в программу;
* обеспечение различного представления данных – с помощью SQL можно представить такую структуру данных, что тот или иной пользователь будет видеть различные их представления. Кроме того, данные из разных частей БД могут быть скомбинированы и представлены в виде одной простой *таблицы*, а значит, представления пригодны для усиления защиты БД и ее настройки под конкретные требования отдельных пользователей;
* возможность динамического изменения и расширения структуры БД – язык SQL позволяет манипулировать структурой БД, тем самым обеспечивая гибкость с точки зрения приспособленности БД к изменяющимся требованиям предметной области;
* поддержка архитектуры *клиент-сервер* – SQL – одно из лучших средств для реализации приложений на платформе *клиент-сервер*. SQL служит связующим звеном между взаимодействующей с пользователем клиентской системой и серверной системой, управляющей БД, позволяя каждой из них сосредоточиться на выполнении своих функций.

Любой язык работы с *базами данных* должен предоставлять пользователю следующие возможности:

* создавать *базы данных* и *таблицы* с полным описанием их структуры;
* выполнять основные операции манипулирования данными, в частности, вставку, модификацию и удаление данных из *таблиц* ;
* выполнять простые и сложные *запросы*, осуществляющие преобразование данных.

Кроме того, язык работы с *базами данных* должен решать все указанные выше задачи при минимальных усилиях со стороны пользователя, а структура и *синтаксис* его команд – достаточно просты и доступны для изучения. И наконец, он должен быть универсальным, т.е. отвечать некоторому признанному *стандарту*, что позволит использовать один и тот же *синтаксис* и структуру команд при переходе от одной *СУБД* к другой. Язык *SQL* удовлетворяет практически всем этим требованиям.

Язык *SQL* является примером языка с трансформирующейся ориентацией, или же языка, предназначенного для работы с *таблицами* с целью преобразования входных данных к требуемому выходному виду. Он включает только команды определения и манипулирования данными и не содержит каких-либо команд управления ходом вычислений. Подобные задачи должны решаться либо с помощью языков программирования или управления заданиями, либо интерактивно, в результате действий, выполняемых самим пользователем. *По*причине подобной незавершенности в плане организации вычислительного процесса язык *SQL* может использоваться двумя способами. Первый предусматривает интерактивную работу, заключающуюся во вводе пользователем с терминала отдельных *SQL*-операторов. Второй состоит во *внедрении SQL-операторов* в программы на *процедурных языках*. Язык *SQL* относительно прост в изучении. Поскольку это не *процедурный язык*, в нем необходимо указывать, какая *информация* должна быть получена, а не как ее можно получить. Иначе говоря, *SQL* не требует указания методов доступа к данным. Как и большинство современных языков, он поддерживает свободный формат записи операторов. Это означает, что при вводе отдельные элементы операторов не связаны с фиксированными позициями экрана. Язык *SQL* может использоваться широким кругом специалистов, включая администраторов *баз данных*, прикладных программистов и множество других конечных пользователей.

Язык *SQL* – первый и пока единственный *стандартный язык* для работы с *базами данных*, который получил достаточно широкое распространение. Практически все крупнейшие разработчики *СУБД* в настоящее время создают свои продукты с использованием языка *SQL* либо с *SQL*-интерфейсом. В него сделаны огромные инвестиции как со стороны разработчиков, так и со стороны пользователей. Он стал частью архитектуры приложений, является стратегическим выбором многих крупных и влиятельных организаций.

Язык *SQL* используется в других стандартах и даже оказывает влияние на разработку иных стандартов как инструмент определения (например, стандарт *Remote* *Data* Access, *RDA*). Создание языка способствовало не только выработке необходимых теоретических основ, но и подготовке успешно реализованных технических решений. Это особенно справедливо в отношении оптимизации *запросов*, методов распределения данных и реализации средств защиты. Начали появляться *специализированные реализации* языка, предназначенные для новых рынков: системы управления обработкой транзакций (OnLine *Transaction* *Processing*, *OLTP* ) и системы оперативной аналитической обработки или системы поддержки *принятия решений* (OnLine Analytical *Processing*, *OLAP* ). Уже известны планы дальнейших расширений *стандарта*, включающих поддержку распределенной обработки, объектно-ориентированного программирования, расширений пользователей и *мультимедиа*.

#### SEQUEL/SQL СУБД System R

Язык для взаимодействия с БД SQL появился в середине 70-х и был разработан в рамках проекта экспериментальной реляционной СУБД System R. Исходное название языка SEQUEL (Structered English Query Language) только частично отражает суть этого языка. Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционной БД, но на самом деле уже являлся полным языком БД, содержащим помимо операторов формулирования запросов и манипулирования БД средства определения и манипулирования схемой БД; определения ограничений целостности и триггеров; представлений БД; возможности определения структур физического уровня, поддерживающих эффективное выполнение запросов; авторизации доступа к отношениям и их полям; точек сохранения транзакции и откатов. В языке отсутствовали средства синхронизации доступа к объектам БД со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет СУБД.

Рассмотрим эти свойства языка немного более подробно.

##### **Запросы и операторы манипулирования данными**

Как известно, двумя фундаментальными языками запросов к реляционным БД являются языки реляционной алгебры и реляционного исчисления. При всей своей строгости и теоретической обоснованности эти языки редко используются в современных реляционных СУБД в качестве средств пользовательского интерфейса. Запросы на этих языках трудно формулировать и понимать. SQL представляет собой некоторую комбинацию реляционного исчисления кортежей и реляционной алгебры, причем до сих пор нет общего согласия, к какому из классических языков он ближе. При этом возможности SQL шире, чем у этих базовых реляционных языков, в частности, в общем случае невозможна трансляция запроса, сформулированного на SQL, в выражение реляционной алгебры, требуется некоторое ее расширение.

Существенными свойствами подъязыка запросов SQL являются возможность простого формулирования запросов с соединениями нескольких отношений и использование вложенных подзапросов в предикатах выборки. Вообще говоря, одновременное наличие обоих средств избыточно, но это дает пользователю при формулировании запроса возможность выбора более понятного ему варианта.

В предикатах со вложенными подзапросами в SQL System R можно употреблять теретико-множественные операторы сравнения, что позволяет формулировать квантифицированные запросы (эти возможности обычно труднее всего понимаются пользователями и поэтому в дальнейшем в SQL появились явно квантифицируемые предикаты).

Существенной особенностью SQL является возможность указания в запросе потребности группирования отношения-результата по указанным полям с поддержкой условий выборки на всю группу целиком. Такие условия выборки могут содержать агрегатные функции, вычисляемые на группе. Эта возможность SQL главным образом отличает этот язык от языков реляционной алгебры и реляционного исчисления, не содержащих аналогичных средств.

Еще одним отличием SQL является необязательное удаление кортежей-дубликатов в окончательном или промежуточных отношениях-результатах. Строго говоря, результатом оператора выборки в языке SQL является не отношение, а мультимножество кортежей. В тех случаях, когда семантика запроса требует наличия отношения, уничтожение дубликатов производится неявно.

Самый общий вид запроса на языке SQL представляет теоретико-множественное алгебраическое выражение, составленное из элементарных запросов. В SQL System R допускались все базовые теретико-множественные операции (UNION, INTERSECT и MINUS).

Работа с неопределенными значениями в SQL System R до конца продумана не была, хотя неявно предполагалось использование трехзначной логики при вычислении логических выражений.

Операторы манипулирования данными UPDATE и DELETE построены на тех же принципах, что и оператор выборки данных SELECT. Набор кортежей указанного отношения, подлежащих модификации или удалению, определяется входящим в соответствующий оператор логическим выражением, которое может включать сложные предикаты, в том числе и с вложенными подзапросами.

В операторе вставки кортежа(ей) в указанное отношение заносимый кортеж может задаваться как в литеральной форме, так и с помощью внутреннего подоператора выборки.

##### Операторы определения и манипулирования схемой БД

В число операторов определения схемы БД SQL System R входили операторы создания и уничтожения постоянных и временных хранимых отношений (CREATE TABLE и DROP TABLE) и создания и уничтожения представляемых отношений (CREATE VIEW и DROP VIEW). В языке и в реализации System R не запрещалось использовать операторы определения схемы в пределах транзакции, содержащей операторы выборки и манипулирования данными. Допускалось, например, использование операторов выборки и манипулирования данными, в которых указываются отношения, не существующие в БД к моменту компиляции оператора. Конечно, эта возможность существенно усложняла реализацию и требовалась по существу очень редко.

Оператор манипулирования схемой БД ALTER TABLE позволял добавлять указываемые поля к существующим отношениям. В описании языка определялось, что выполнение этого оператора не должно приводить к недействительности ранее откомпилированных операторов над отношением, схема которого изменяется, и что значения вновь определенных полей в существующих кортежах отношения становятся неопределенными.

##### Определения ограничений целостности и триггеров

Язык SQL System R включал очень мощные средства контроля и поддержания целостности БД. Средства контроля базировались на аппарате ограничений целостности (ASSERTIONS). Фактически, ограничение целостности - это логическое выражение, вычисляемое над текущим состоянием БД, ложность которого соответствует нецелостному состоянию БД. Логическое выражение ограничения целостности могло содержать любой допустимый в языке предикат.

Более точно, ограничения целостности делились на два класса: проверяемые после выполнения оператора манипулирования данными и проверяемые при завершении транзакции или при выполнении специального оператора INFORCE INTEGRITY. Типы предикатов, которые можно использовать в операторах определения ограничений целостности разных классов, различаются. В операторах первого класса проверяется, фактически, текущий кортеж, с которым производится манипулирование. Во втором случае проверяются указанные в ограничении целостности отношения, т.е. все их кортежи. Различается и определяемая в языке реакция системы на нарушения ограничений целостности разных классов. В первом случае нарушение ограничения целостности приводит к откату транзакции в точку, непосредственно предшествующую операции манипулирования данными, выполнение которого вызвало нарушение ограничения целостности. Во втором случае ограничение приводит к полному откату транзакции к ее началу.

Очень важным механизмом, определенным в языке SQL System R, является механизм триггеров. В контексте System R этот механизм рассматривался главным образом как средство автоматического поддержания целостности БД. При определении триггера указывалось условие проверки его применимости (имя отношения и тип операции манипулирования данными), условие применимости триггера (логическое выражение, построенное по правилам, близким к правилам для ограничений целостности первого класса) и действие, которое должно быть выполнено над БД в случае истинности условия применимости. Такое действие могло быть выражено с помощью произвольного оператора манипулирования данными. Во время выполнения действия могли срабатывать другие триггеры и т.д.

Механизмы ограничений целостности и триггеров System R являлись очень мощными и общими, но реализация их очень трудна и накладна (как уже отмечалось, триггеры так и не были реализованы в System R). Дополнительную сложность в реализации создавал тот факт, что допускалось (по крайней мере не запрещалось языком) определение ограничений целостности и триггеров в пределах той же транзакции, в которой выполняются операторы манипулирования данными. При наиболее полной реализации требовалось бы большое число дополнительных действий во время выполнения транзакции. Кроме того, в ряде случаев отсутствие зафиксированной семантики соответствующих конструкций языка приводило к неоднозначному пониманию выполнения транзакций.

##### Представления базы данных

В языке допускалось использование хранимых отношений БД и представляемых отношений. Наиболее удачным решением было использование для определения представлений общего аппарата операторов выборки. Любой оператор выборки может быть использован для определения представления.

В языке отсутствуют какие-либо ограничения по поводу использования представлений: в любом операторе SQL, в котором допускается использование имени хранимого отношения, допускается и использование имени представления. В SQL System R ничего не говорится о рекомендуемом способе реализации доступа к представлениям, но при любом способе эффект должен быть таким, как если бы выполнить полную материализацию представления до выполнения оператора.

Массу проблем, исследований и предложений породила потенциальная возможность выполнения операторов манипулирования данными над представлениями. Понятно, что эта возможность легко реализуема для простых представлений, но в более сложных случаях не только реализация, но и семантика операций становится нетривиальной. Кстати, в System R операторы манипулирования данными допускались только над простыми представлениями.

##### Определение управляющих структур

Внесение в реляционный язык, каким является SQL, явных операторов порождения и уничтожения структур физического уровня, поддерживающих эффективное выполнение запросов к БД, явилось в SQL System R чисто прагматическим решением, обеспечивающим возможность всех видов работ с БД с помощью одного языка.

В SQL System R упоминаются два вида таких структур: индексы и связи (links). Индекс в его абстрактном языковом представлении - это инвертированный файл, обеспечивающий доступ к кортежам соответствующего отношения на основе заданных значений одного или нескольких столбцов, составляющих ключ индекса. Операторы языка позволяли создавать и уничтожать индексы, но никаким образом не давали возможности явно указать на необходимость использования существующего индекса при выполнении оператора выборки, решение об этом возлагалось на реализацию.

С помощью оператора определения индекса можно было выразить два дополнительных утверждения, касающихся логической схемы отношения и физической структуры его хранения. Использование при определении индекса ключевого слова UNIQUE означало, что ключ этого индекса является возможным ключом соответствующего отношения. Фактически это означает наличие дополнительного механизма определения ограничения целостности отношения. Один из индексов для данного отношения мог быть определен с ключевым словом CLUSTERING. Это означает требование физической кластеризации во внешней памяти кортежей отношения с равными или близкими значениями ключа индекса.

Операторы определения связи позволяли в стиле сетевой модели данных организовать во внешней памяти списки кортежей указанного отношения. Как и в случае индексов, операторы позволяли создавать и уничтожать такие списки, но не давали возможности явно указать на необходимость использования существующих списков при выполнении операторов выборки. Большая трудоемкость поддержания списков при выполнении операторов манипулирования данными и трудность выполнения оценок стоимости их использования при выполнении операторов выборки привели к тому, что механизм связей исчез из языка уже на поздней стадии проекта System R. С тех пор этот механизм, насколько нам известно, не появлялся ни в одном варианте SQL.

##### Авторизация доступа к отношениям и их полям

Существенной особенностью языка SQL, появившейся в нем с самого начала, является обеспечение защиты доступа к данным средствами самого языка. Основная идея такого подхода состоит в том, что по отношению к любому отношению БД и любому столбцу отношения вводится предопределенный набор привилегий. С каждой транзакцией неявно связывается идентификатор пользователя, от имени которого она выполняется (способы связи и идентификации пользователей не фиксируются в языке и определяются в реализации).

После создания нового отношения все привилегии, связанные с этим отношением и всеми его столбцами, принадлежат только пользователю-создателю отношения. В число привилегий входит привилегия передачи всех или части привилегий другому пользователю, включая привилегию на передачу привилегий. Технически передача привилегий осуществляется при выполнении оператора SQL GRANT. Существует также привилегия изъятия всех или части привилегий у пользователя, которому они ранее были переданы. Эта привилегия также может передаваться. Технически изъятие привилегий происходит при выполнении оператора SQL REVOKE.

Проверка полномочности доступа к данным происходит на основе информации о полномочиях, существующих во время компиляции соответствующего оператора SQL. Подобно тому, что мы отмечали в связи с ограничениями целостности и триггерами, в SQL System R отсутствовали какие-либо ограничения по поводу использования операторов GRANT и REVOKE. Это приводило к существенным техническим затруднениям в реализации, а иногда к неоднозначному пониманию поведения.

Долгое время подход к защите данных от несанкционированного доступа принимался практически без критики, однако в связи с распространяющимся использованием реляционных СУБД в нетрадиционных приложениях все чаще раздается критика. Если, например, в системе БД должна поддерживаться многоуровневая защита данных, соответствующую систему полномочий весьма трудно, а иногда и невозможно построить на основе средств SQL.

##### Точки сохранения и откаты транзакции

В SQL System R существовали два специальных оператора для установки так называемых точек сохранения транзакции и для отката транзакции к ранее установленной точке сохранения. В литературе, относящейся к System R, обсуждение этих возможностей практически не содержится, из чего неявно следует, что они не были реализованы.

Прямолинейная реализация этого механизма не вызывает особых технических затруднений, но и не очень полезна, потому что после выполнения частичного отката транзакции для успешного продолжения работы прикладной программы потребовалось бы и восстановить ее состояние в соответствующей точке, а это никак не поддерживается. Понятно, что при более тщательной проработке должны быть увязаны механизмы точек сохранения и контроля целостности. Например, было бы естественно, чтобы при выполнении оператора ENFORCE INTEGRITY, если какие-либо ограничения целостности нарушаются, происходил автоматический откат транзакции к ближайшей точки сохранения, в которой нарушения целостности БД не было. Это значительно усложнило бы реализацию, но было бы очень полезно. Аналогично, можно было бы использовать механизм точек сохранения при автоматических откатах транзакций по причине возникновения синхронизационных тупиков.

Отметим еще два важных свойства языка SQL System R, которые в разных видах присутствуют во всех развитых последующих вариантах языка.

##### Встроенный SQL

В SQL System R присутствуют специальные операторы, поддерживающие встраивание операторов SQL в традиционные языки программирования (в System R основным таким языком был PL/1).

Основная проблема встраивания SQL в язык программирования состояла в том, что SQL - реляционный язык, т.е. его операторы большей частью работают со множествами, в то время как в языках программирования основными являются скалярные операции. Решение SQL состоит в том, что в язык дополнительно включаются операторы, обеспечивающие покортежный доступ к результату запроса к БД.

Для этого в язык вводится понятие курсора, с которым связывается оператор выборки. Над определенным курсором можно выполнять оператор OPEN, означающий материализацию отношения-результата запроса, оператор FETCH, позволяющий выбрать очередной кортеж результирующего отношения в память программы, и оператор CLOSE, означающий конец работы с данным курсором.

Дополнительную гибкость при создании прикладных программ со встроенным SQL обеспечивает возможность параметризации операторов SQL значениями переменных включающей программы.

##### Динамический SQL

Для упрощения создания интерактивных SQL-ориентированных систем в SQL System R были включены операторы, позволяющие во время выполнения транзакции откомпилировать и выполнить любой оператор SQL.

Оператор PREPARE вызывает динамическую компиляцию оператора SQL, текст которого содержится в указанной переменной символьной строке включающей программы. Текст может быть помещен в переменную при выполнении программы любым допустимым способом, например, введен с терминала.

Оператор DESCRIBE служит для получения информации об указанном операторе SQL, ранее подготовленном с помощью оператора PREPARE. C помощью этого оператора можно узнать, во-первых, является ли подготовленный оператор оператором выборки, и во-вторых, если это оператор выборки, получить полную информацию о числе и типах столбцов результирующего отношения.

Для выполнения ранее подготовленного оператора SQL, не являющегося оператором выборки, служит оператор EXECUTE. Для выполнения динамически подготовленного оператора выборки используется аппарат курсоров с некоторыми отличиями по части задания адресов переменных включающей программы, в которые должны быть помещены значения столбцов текущего кортежа результата.

Подводя итог приведенному краткому описанию основных черт SQL System R, отметим, что несмотря на недостаточную техническую проработку, в идейном отношении язык содержал все необходимые средства, позволяющие использовать его как базовый язык СУБД.

## 15. Журнализация изменений БД. Журнализация и буферизация.

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

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

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

Проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое оперативной памяти не утрачено и можно пользоваться содержимым как буфера журнала, так и буферов страниц базы данных. Но если произошел мягкий сбой, и содержимое буферов утрачено, для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц базы данных является то, что запись об изменении объекта базы данных должна попадать во внешнюю память журнала раньше, чем измененный объект оказывается во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется Write Ahead Log (WAL) - "пиши сначала в журнал", и состоит в том, что если требуется вытолкнуть во внешнюю память измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала записи о его изменении.

Другими словами, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. если во внешней памяти журнале содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершившаяся транзакция должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние базы данных, содержащее результаты всех зафиксированных к моменту сбоя транзакций.

Простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц базы данных, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные расходы при выполнении операции фиксации транзакции.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная запись о конце транзакции.

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

## 16. Восстановление баз данных в различных ситуациях.

#### Индивидуальный откат транзакции

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

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

* Выбирается очередная запись из списка данной транзакции.
* Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT, и вместо прямой операции UPDATE обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.
* Любая из этих обратных операций также журнализуются. Собственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.
* При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

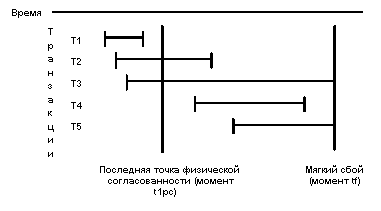
#### Восстановление после мягкого сбоя

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

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

Будем считать, что в журнале отмечаются точки физической согласованности базы данных - моменты времени, в 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. Немного позже мы рассмотрим, как можно достичь физической согласованности. Назовем такие точки tpc (time of physical consistency).

Тогда к моменту мягкого сбоя возможны следующие состояния транзакций:



Предположим, что некоторым способом удалось восстановить внешнюю память базы данных к состоянию на момент времени tlpc (как это можно сделать - немного позже). Тогда:

* Для транзакции T1 никаких действий производить не требуется. Она закончилась до момента tlpc, и все ее результаты отражены во внешней памяти базы данных.
* Для транзакции T2 нужно повторно выполнить оставшуюся часть операций (redo). Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T2 после момента tlpc. Следовательно, повторная прямая интерпретация операций T2 корректна и приведет к логически согласованному состоянию базы данных (поскольку транзакция T2 успешно завершилась до момента мягкого сбоя, в журнале содержатся записи обо всех изменениях, произведенных этой транзакцией).
* Для транзакции T3 нужно выполнить в обратном направлении первую часть операций (undo). Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T3, которые были выполнены после момента tlpc. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T3, которые были выполнены до момента tlpc. Следовательно, обратная интерпретация операций T3 корректна и приведет к согласованному состоянию базы данных (поскольку транзакция T3 не завершилась к моменту мягкого сбоя, при восстановлении необходимо устранить все последствия ее выполнения).
* Для транзакции T4, которая успела начаться после момента tlpc и закончиться до момента мягкого сбоя, нужно выполнить полную повторную прямую интерпретацию операций (redo).
* Наконец, для начавшейся после момента tlpc и не успевшей завершиться к моменту мягкого сбоя транзакции T5 никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

## 17. Язык SQL, функции и основные возможности. Стандартизация SQL.

## 18. Типы данных языка SQL. Средства определения схемы.

**Типы данных SQL** разделяются на три группы:  
— **строковые**;  
— **с плавающей точкой** (**дробные числа**);  
— **целые числа**, **дата** и **время**.

1. **Типы данных SQL** строковые

|  |  |
| --- | --- |
| **Типы данных SQL** | **Описание** |
| CHAR(size) | Строки фиксированной длиной (могут содержать буквы, цифры и специальные символы). Фиксированный размер указан в скобках. Можно записать до 255 символов |
| VARCHAR(size) | Может хранить не более 255 символов. |
| TINYTEXT | Может хранить не более 255 символов. |
| TEXT | Может хранить не более 65 535 символов. |
| BLOB | Может хранить не более 65 535 символов. |
| MEDIUMTEXT | Может хранить не более 16 777 215 символов. |
| MEDIUMBLOB | Может хранить не более 16 777 215 символов. |
| LONGTEXT | Может хранить не более 4 294 967 295 символов. |
| LONGBLOB | Может хранить не более 4 294 967 295 символов. |
| ENUM(x,y,z,etc.) | Позволяет вводить список допустимых значений. Можно ввести до 65535 значений в **SQL Тип данных ENUM** список. Если при вставке значения не будет присутствовать в списке ENUM, то мы получим пустое значение. Ввести возможные значения можно в таком формате: ENUM ( 'X', 'Y', 'Z') |
| SET | **SQL Тип данных SET** напоминает ENUM за исключением того, что SET может содержать до 64 значений. |

1. **Типы данных SQL** с плавающей точкой (дробные числа) и целые числа

|  |  |
| --- | --- |
| **Типы данных SQL** | **Описание** |
| TINYINT(size) | Может хранить числа от -128 до 127 |
| SMALLINT(size) | Диапазон от -32 768 до 32 767 |
| MEDIUMINT(size) | Диапазон от -8 388 608 до 8 388 607 |
| INT(size) | Диапазон от -2 147 483 648 до 2 147 483 647 |
| BIGINT(size) | Диапазон от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 |
| FLOAT(size,d) | Число с плавающей точкой небольшой точности. |
| DOUBLE(size,d) | Число с плавающей точкой двойной точности. |
| DECIMAL(size,d) | Дробное число, хранящееся в виде строки. |

1. **Типы данных SQL** — Дата и время

|  |  |
| --- | --- |
| **Типы данных SQL** | **Описание** |
| DATE() | Дата в формате ГГГГ-ММ-ДД |
| DATETIME() | Дата и время в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС |
| TIMESTAMP() | Дата и время в формате timestamp. Однако при получении значения поля оно отображается не в формате timestamp, а в виде ГГГГ-ММ-ДД ЧЧ:ММ:СС |
| TIME() | Время в формате ЧЧ:ММ:СС |
| YEAR() | Год в двух значной или в четырехзначном формате. |

#### 

#### Средства определения схемы

Средства определения схемы БД в стандарте SQL/89 относятся к наиболее слабым и допускающим различную интерпретацию частям стандарта. Более того, мне неизвестна ни одна реализация, в которой поддерживался бы в точности такой набор средств определения схемы.

Поэтому, чтобы добиться мобильности прикладной системы в достаточно широком классе реализаций SQL/89, необходимо тщательно локализовать компоненты определения схемы БД. Думаю, что лучше всего сосредоточить всю работу со схемой БД в одном модуле и иметь в виду, что при переходе к другой СУБД очень вероятно потребуется переделка этого модуля.

Особо отметим, что в SQL/89 вообще отсутствуют какие-либо средства изменения схемы БД: нет возможности удалить схему таблицы, добавить к схеме таблицы новый столбец и т.д. Во всех реализациях такие средства поддерживаются, но они могут различаться и синтаксисом, и семантикой.

Несмотря на отсутствие особых надежд на то, что удастся встретить реализацию, поддерживающую язык определения схем SQL/89, мы коротко опишем этот язык (без синтаксических деталей), чтобы оценить на содержательном уровне возможности SQL/89 в этой части и получить хотя бы какие-то средства сравнения разных реализаций.

##### Оператор определения схемы

В соответствии с правилами SQL/89 каждая таблица данной БД имеет простое и квалифицированное имена. В качестве квалификатора имени выступает "идентификатор полномочий" таблицы, который обычно в реализациях совпадает с именем некоторого пользователя, и квалифицированное имя таблицы имеет вид:

<идентификатор полномочий>.<простое имя>

Подход к определению схемы в SQL/89 состоит в том, что все таблицы с одним идентификатором полномочий создаются (определяются) путем выполнения одного оператора определения схемы. При этом в стандарте не определяется способ выполнения оператора определения схемы: должен ли он выполняться только в интерактивном режиме или может быть встроен в программу, написанную на традиционном языке программирования.

В операторе определения схемы содержится идентификатор полномочий и список элементов схемы, каждый из которых может быть определением таблицы, определением представления (view) или определением привилегий. Каждое из этих определений представляется отдельным оператором SQL/89, но все они, как уже говорилось, должны быть встроены в оператор определения схемы.

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

##### Определение таблицы

Оператор определения таблицы имеет следующий синтаксис:

<table definition> ::=

CREATE TABLE <table name> (<table element>

[{,<table element>}...])

<table element> ::=

<column definition>

| <table constraint definition>

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

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

##### Определение столбца

Оператор определения столбца описывается следующими синтаксическими правилами:

<column definition> ::=

<column name> <data type>

[<default clause>] [<column constraint>...]

<default clause> ::=

DEFAULT { <literal> | USER | NULL }

<column constraint> ::=

NOT NULL [<unique specification>]

| <references specification>

| CHECK (<search condition>)

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

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

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

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

FOREIGN KEY(C) <reference specification>.

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

##### Определение ограничений целостности таблицы

Ограничения целостности таблицы обладают следующим синтаксисом:

<table constraint definition> ::=

<unique constraint definition>

| <referential constraint definition>

| <check constraint definition>

<unique constraint definition> ::=

<unique specification> (<unique column list>)

<unique specification> ::= UNIQUE | PRIMARY KEY

<unique column list> ::= <column name> [{,<column name>}...]

<referential constraint definition> ::=

FOREIGN KEY (<referencing columns>) <references specification>

<references specification> ::=

REFERENCES <referenced table and columns>

<referencing columns> ::= <reference column list>

<referenced table and columns> ::=

<table name> [(<reference column list>)]

<reference column list> ::= <column name> [{,<column name>}...]

<check constraint definition> ::= CHECK (<search condition>)

Для одной таблицы может быть задано несколько ограничений целостности, в том числе те, которые неявно порождаются ограничениями целостности столбцов. Стандарт SQL/89 устанавливает, что ограничения таблицы фактически проверяются при выполнении каждого оператора SQL.

Замечание: Наличие правильно подобранного набора ограничений БД очень важно для надежного функционирования прикладной информационной системы. Вместе с тем, в некоторых СУБД ограничения целостности практически не поддерживаются. Поэтому при проектировании прикладной системы необходимо принять решение о том, что более существенно: рассчитывать на поддержку ограничений целостности, но ограничить набор возможных СУБД, или отказаться от их использования на уровне SQL, сохранив возможность использования не самых современных СУБД.

Далее T обозначает таблицу, для которой определяются ограничения целостности.

**Ограничение уникальности**

Каждое имя столбца в списке уникальности должно именовать столбец T и не должно входить в этот список более одного раза. При определении столбца, входящего в список уникальности, должно быть указано ограничение столбца NO NULL. Среди ограничений уникальности T не должно быть более одного определения первичного ключа (ограничения уникальности с ключевым словом RIMARY KEY).

Действие ограничения уникальности состоит в том, что в таблице T не допускается появление двух или более строк, значения столбцов уникальности которых совпадают.

**Ограничение по ссылкам**

Ограничение по ссылкам от заданного набора столбцов CT таблицы T на заданный набор столбцов CT1 некоторой определенной к этому моменту таблицы T1 определяет условие на содержимое обеих этих таблиц, при котором ссылки можно считать корректными.

Если список столбцов CT1 явно специфицирован в определении ограничения по ссылкам, то требуется, чтобы этот список явно входил в какое-либо определение уникальности таблицы T1. Если же список CT1 не специфицирован явно в определении ограничения по ссылкам таблицы T, то требуется, чтобы в определении таблицы T1 присутствовало определение первичного ключа, и список CT1 неявно полагается совпадающим со списком имен столбцов из определения первичного ключа таблицы T1. Имена столбцов списков CT и CT1 должны именовать столбцы таблиц T и T1 соответственно, и не должны появляться в списках более одного раза. Списки столбцов CT и CT1 должны содержать одинаковое число элементов, и столбец таблицы T, идентифицируемый i-ым элементом списка CT должен иметь тот же тип, что столбец таблицы T1, идентифицируемый i-ым элементом списка CT1.

По определению, таблицы T и T1 удовлетворяют заданному ограничению по ссылкам, если для каждой строки s таблицы T такой, что все значения столбцов s, идентифицируемых списком CT, не являются неопределенными, существует строка s1 таблицы T1 такая, что значения столбцов s1, идентифицируемых списком CT1, позиционно равны значениям столбцов s, идентифицируемых списком CT. По человечески это можно сформулировать так: ограничение по ссылкам удовлетворяется, если для каждой корректной ссылки существует объект, на который она ссылается. В привычной программистам терминологии, ограничение по ссылкам не позволяет производить "висячие" ссылки, не ведущие ни к какому объекту.

**Проверочное ограничение**

Проверочное ограничение специфицирует условие, которому должна удовлетворять в отдельности каждая строка таблицы T. Это условие не должно содержать подзапросов, спецификаций агрегатных функций, а также ссылок на внешние переменные или параметров. В него могут входить только имена столбцов данной таблицы и литеральные константы.

Таблица удовлетворяет проверочному ограничению целостности в том и только в том случае, когда вычисление условия для каждой строки таблицы дает true.

Замечание: В некоторых реализациях допускаются расширенные механизмы ограничений по ссылкам и проверочных ограничений. Следует быть внимательным, если не желать выходить за пределы возможностей стандарта.

##### Определение представлений

Механизм представлений (view) является мощным средством языка SQL, позволяющим скрыть реальную структуру БД от некоторых пользователей за счет определения представления БД, которое реально является некоторым хранимым в БД запросом с именованными столбцами, а для пользователя ничем не отличается от базовой таблицы БД (с учетом технических ограничений). Любая реализация должна гарантировать, что состояние представляемой таблицы точно соответствует состоянию базовых таблиц, на которых определено представление. Обычно вычисление представляемой таблицы (материализация соответствующего запроса) производится каждый раз при использовании представления.

В стандарте SQL/89 оператор определения представления имеет следующий синтаксис:

<view definition> ::=

CREATE VIEW <table name> [(<view column list>)]

AS <query specification> [WITH CHECK OPTION]

<view column list> ::= <column name> [{,<column name>}...]

Определяемая представляемая таблица V является изменяемой (т.е. по отношению к V можно использовать операторы DELETE и UPDATE) в том и только в том случае, если выполняются следующие условия для спецификации запроса:

* В списке выборки не указано ключевое слово DISTINCT;
* Каждое арифметическое выражение в списке выборки представляет собой одну спецификацию столбца, и спецификация одного столбца не появляется более одного раза;
* В разделе FROM указана только одна таблица, являющаяся либо базовой таблицей, либо изменяемой представляемой таблицей;
* В условии выборки раздела WHERE не используются подзапросы;
* В табличном выражении отсутствуют разделы GROUP BY и HAVING.

Замечание: Эти ограничения являются очень сильными. В реализациях они могут быть ослаблены. Но если стремиться к мобильности, не следует пользоваться расширенными возможностями.

Если в списке выборки спецификации запроса имеется хотя бы одно арифметическое выражение, состоящее не из одной спецификации столбца, или если одно имя столбца участвует в списке выборки более одного раза, определение представления должно содержать список имен столбцов представляемой таблицы. Более просто, нужно явно именовать столбцы представляемой таблицы, если эти имена не наследуются от столбцов таблиц раздела FROM спецификации запроса.

Требование WITH CHECK OPTION в определении представления имеет смысл только в случае определения изменяемой представляемой таблицы, которая определяется спецификацией запроса, содержащей раздел WHERE. При наличии этого требования не допускаются изменения представляемой таблицы, которые приводят к появлению в базовых таблиц строк, не видимых в представляемой таблице (т.е. таких строк, которые не удовлетворяют условию поиска раздела WHERE спецификации запроса). Если WITH CHECK OPTION в определении представления отсутствует, такой контроль не производится.

##### Определение привилегий

В соответствии с идеологией языка SQL контроль прав доступа данного пользователя к таблицам БД производится на основе механизма привилегий. Фактически, этот механизм состоит в том, что для выполнения любого действия над таблицей пользователь должен обладать соответствующей привилегией (реально все возможные действия описываются фиксированным стандартным набором привилегий). Пользователь, создавший таблицу, автоматически становится владельцем всех возможных привилегий на выполнение операций над этой таблицей. В число этих привилегий входит привилегия на передачу всех или некоторых привилегий по отношению к данной таблице другому пользователю, включая привилегию на передачу привилегий. Иногда поддерживается и обратная операция изъятия привилегий от пользователя, ранее их получившего.

В SQL/89 определяется упрощенная схема механизма привилегий. Во-первых, "раздача" привилегий возможна только при определении таблицы. Во-вторых, пользователь, получивший некоторые привилегии от других пользователей, может передать их дальше только при определении схемы.

Определение привилегий производится в следующем синтаксисе:

<privilege definition> ::=

GRANT <privileges> ON <table name> TO <grantee>

[{,<grantee>}...] [WITH GRANT OPTION]

<privileges> ::=

ALL PRIVILEGES

| <action> [{,<action>}...]

<action> ::= SELECT | INSERT | DELETE

| UPDATE [(<grant column list>)]

| REFERENCES [(<grant column list>]

<grant column list> ::= <column name> [{,<column name>}...]

<grantee> ::= PUBLIC | <authorization identifier>

Смысл механизма определения привилегий в SQL/89 достаточно понятен из этого синтаксиса. Заметим лишь, что привилегией REFERENCES по отношению к указанным столбцам таблицы T1 необходимо обладать, чтобы иметь возможность при определении таблицы T определить ограничение по ссылкам между этой таблицей и существующей к этому моменту таблицей T1.

Еще раз заметим, что хотя в общем смысле во всех SQL-ориентированных СУБД поддерживается механизм защиты доступа на основе привилегий, реализации могут различаться в деталях. Это опять то место, которое нужно локализовывать, если стремиться к созданию мобильной прикладной системы.

## 19. Язык SQL, средства манипулирования данными. Структура запросов.

#### Структура запросов

Для того, чтобы можно было более или менее точно рассказать про структуру запросов в стандарте SQL/89, необходимо начать со сводки синтаксических правил:

<cursor specification> ::=

<query expression> [<order by clause>]

<query expression> ::=

<query term>

| <query expression> UNION [ALL] <query term>

<query term> ::=

<query specification>

| (<query expression>)

<query specification> ::=

(SELECT [ALL | DISTINCT] <select list> <table expression>)

<select statement> ::=

SELECT [ALL | DISTINCT] <select list>

INTO <select target list> <table expression>

<subquery> ::=

(SELECT [ALL | DISTINCT] <result specification>

<table expression>

<table expression> ::=

<from clause>

[<where clause>]

[<group by clause>]

[<having clause>]

Язык допускает три типа синтаксических конструкций, начинающихся с ключевого слова SELECT: спецификация курсора (cursor specification), оператор выборки (select statement) и подзапрос (subquery). Основой всех них является синтаксическая конструкция "табличное выражение (table expression)". Семантика табличного выражения состоит в том, что на основе последовательного применения разделов from, where, group by и having из заданных в разделе from таблиц строится некоторая новая результирующая таблица, порядок следования строк которой не определен и среди строк которой могут находиться дубликаты (т.е. в общем случае таблица-результат табличного выражения является мультимножеством строк). На самом деле именно структура табличного выражения наибольшим образом характеризует структуру запросов языка SQL/89. Мы рассмотрим ниже структуру и смысл разделов табличного выражения ниже, но до этого немного подробнее обсудим три упомянутые конструкции, включающие табличные выражения.

##### Спецификация курсора

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

**Спецификация запроса**

В спецификации запроса задается список выборки (список арифметических выражений над значениями столбцов результата табличного выражения и констант). В результате применения списка выборки к результату табличного выражения производится построение новой таблицы, содержащей то же число строк, но вообще говоря другое число столбцов, содержащих результаты вычисления соответствующих арифметических выражений из списка выборки. Кроме того, в спецификации запроса могут содержаться ключевые слова ALL или DISTINCT. При наличии ключевого слова DISTINCT из таблицы, полученной применением списка выборки к результату табличного выражения, удаляются строки-дубликаты; при указании ALL (или просто при отсутствии DISTINCT) удаление строк-дубликатов не производится.

**Выражение запросов**

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

**Раздел ORDER BY**

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

<order by clause> ::=

ORDER BY <sort specification>

[{,<sort specification>}...]

<sort specification> ::=

{<unsigned integer> | <column specification>}

[ASC | DESC]

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

##### Оператор выборки

Оператор выборки - это отдельный оператор языка SQL/89, позволяющий получить результат запроса в прикладной программе без привлечения курсора. Поэтому оператор выборки имеет синтаксис, отличающийся от синтаксиса спецификации курсора, и при его выполнении возникают ограничения на результат табличного выражения. Фактически, и то, и другое диктуется спецификой оператора выборки как одиночного оператора SQL: при его выполнении результат должен быть помещен в переменные прикладной программы. Поэтому в операторе появляется раздел INTO, содержащий список переменных прикладной программы, и возникает то ограничение, что результирующая таблица должна содержать не более одной строки. Соответственно, результат базового табличного выражения должен содержать не более одной строки, если оператор выборки не содержит спецификации DISTINCT, и таблица, полученная применением списка выборки к результату табличного выражения, не должна содержать более одной несовпадающих строк, если спецификация DISTINCT задана.

Замечание: В диалекте SQL СУБД Oracle поддерживается расширенный вариант оператора выборки, результатом которого не обязательно является таблица из одной строки. Такое расширение не поддерживается ни в SQL/89, ни в SQL/92.

##### Подзапрос

Наконец, последняя конструкция SQL/89, которая может содержать табличные выражения, - это подзапрос, т.е. запрос, который может входить в предикат условия выборки оператора SQL. В SQL/89 к подзапросам применяется то ограничение, что результирующая таблица должна содержать в точности один столбец. Поэтому в синтаксических правилах, определяющих подзапрос, вместо списка выборки указано "выражение, вычисляющее значение", т.е. арифметическое выражение. Заметим еще, что поскольку подзапрос всегда вложен в некоторый другой оператор SQL, то в качестве констант в арифметическом выражении выборки и логических выражениях разделов WHERE и HAVING можно использовать значения столбцов текущих строк таблиц, участвующих в (под)запросах более внешнего уровня. Более подробно об этом см. ниже, при описании семантики табличных выражений.

## 20. Использование SQL при прикладном программировании.

В стандарте SQL/89 определены два способа взаимодействия с БД из прикладной программы, написанной на традиционном языке программирования (как мы уже упоминали, SQL/89 ориентирован на использование совместно с языками Кобол, Фортран, Паскаль и ПЛ/1, но в реализациях обычно поддерживается и язык Си). Первый способ состоит в том, что все операторы SQL, с которыми может работать данная прикладная программа, собраны в один модуль и оформлены как процедуры этого модуля. Для этого SQL/89 содержит специальный подъязык - язык модулей. При использовании такого способа взаимодействия с БД прикладная программа содержит вызовы процедур модуля SQL с передачей им фактических параметров и получением ответных параметров.

Второй способ состоит в использовании так называемого встроенного SQL, когда с использованием специального синтаксиса в программу на традиционном языке программирования встраиваются операторы SQL. В этом случае с точки зрения прикладной программы оператор SQL выполняется "по месту". Явная параметризация операторов SQL отсутствует, но во встроенных операторах SQL могут использоваться имена переменных основной программы, и за счет этого обеспечивается связь между прикладной программой и СУБД.

Концептуально эти два способа эквивалентны. Более того, в стандарте устанавливаются правила порождения неявного модуля SQL по программе со встроенным SQL. Однако в большинстве реализаций операторы SQL, содержащиеся в модуле SQL, и встроенные операторы SQL обрабатываются существенно по-разному. Модуль SQL обычно компилируется отдельно от прикладной программы, в результате чего порождается набор так называемых хранимых процедур (в стандарте этот термин не используется, но распространен в коммерческих реализациях). Т.е. в случае использования модуля SQL компиляция операторов SQL производится один раз, и затем соответствующие процедуры сколько угодно раз могут вызываться из прикладной программы.

В отличие от этого, для операторов SQL, встроенных в прикладную программу, компиляция этих операторов обычно производится каждый раз при их использовании (правильнее сказать, при каждом первом использовании оператора при данном запуске прикладной программы).

Конечно, пользователи не обязаны знать об этом техническом различии в обработке двух видов взаимодействия с СУБД. Существуют и такие системы, которые производят одноразовую компиляцию встроенных операторов SQL и сохраняют откомпилированный код. Но все-таки лучше иметь это в виду.

Приведем некоторые соображения за и против каждого из этих двух способов. При использовании языка модулей текст прикладной программы имеет меньший размер, взаимодействия с СУБД более локализованы за счет наличия явных параметров вызова процедур. С другой стороны, для понимания смысла поведения прикладной программы потребуется одновременное чтение двух текстов. Кроме того, как кажется, синтаксис модуля SQL может существенно различаться в разных реализациях. Встроенный SQL предоставляет возможность производства более "самосодержащихся" прикладных программ. Имеется больше оснований рассчитывать на простоту переноса такой программы в среду другой СУБД, поскольку стандарт встраивания более или менее соблюдается. Основным недостатком является некоторый PL-подобный вид таких программ, независимо от выбранного основного языка. И конечно, нужно учитывать замечания, содержащиеся в предыдущих абзацах.

Далее мы коротко опишем язык модулей и правила встраивания в соответствии со стандартом SQL/89 (еще раз заметим, что формально правила встраивания не являются частью стандарта).

## 21. Проблемы оптимизации. Схема обработки запроса.

**Оптимизация в распределенных системах управления базами данных**

С развитием аппаратуры и программного обеспечения сетей ЭВМ (локальных и территориальных) проблема интеграции разобщенных баз данных становится все более актуальной. Существует много подходов к решению этой проблемы, начиная от обеспечения удаленного сетевого доступа к локальным базам данных и кончая построением сложных распределенных систем управления базами данных, обеспечивающими для пользователя прозрачность местоположения интересующих его данных. В последнем случае пользователи взаимодействуют с глобальной распределенной базой данных на основе ее глобальной схемы.

В этой работе мы не сможем привести полный анализ проблем оптимизации запросов и структур данных и стратегий выполнения реляционных операций во всем спектре распределенных систем баз данных. В этом мы ограничены и объемом данной статьи, и сложностью задачи. Тем не менее, основные особенности оптимизации в некотором широком классе распределенных систем баз данных мы постараемся рассмотреть. К этому классу мы отнесем распределенные системы управления базами данных, основанные на однородных локальных системах управления базами данных. Такой системой является, например, распределенная СУБД System R\* [93].

Дополнительным требованием в System R\*, довольно естественным, как нам кажется, для систем этого класса, является требование локальной автономности узлов сети, на которых выполняются локальные представители СУБД (незначительно модифицированная System R). Это требование означает отсутствие централизованного администрирования глобальной базой данных, отсутствие централизованного каталога базы данных и т.д. Должно допускаться локальное порождение и удаление новых отношений и индексов в локальной базе данных, причем уничтожение индекса не должно приводить к невозможности повторного выполнения ранее откомпилированных глобальных запросов. Требование локальной автономности существенно влияет и на допустимые способы оптимизации глобальных запросов (здесь и далее под глобальным запросом мы понимаем запрос, для выполнения которого требуется сетевой доступ к удаленным локальным базам данных).

Заметим, что логические уровни оптимизации запросов, включая семантическую оптимизацию, фактически, не связаны с распределенной или локальной природой баз данных (не считая некоторой специфики обработки запросов через представления). Распределенный характер баз данных затрагивает главным образом "физические" уровни оптимизации, связанные с выбором и оценкой планов выполнения запроса.

Другая проблема распределенных систем баз данных касается алгоритмов выполнения двуместных реляционных операций над отношениями, хранящимися в разных узлах распределенной системы. Эта проблема существует и при традиционном хранении каждого отношения целиком в одной локальной базе данных, но становится еще более сложной и допускающей большее число решений в случае так называемых разделенных (partitioned) и раскопированных (replicated) баз данных. Такие нетрадиционные организации распределенных баз данных очень активно обсуждаются в литературе, хотя нам неизвестна какая-либо завершенная система, построенная на этих принципах. (В свое время в начальном проекте System R\* предполагалось использование таких организаций, но ни одна из них не была практически реализована).

#### Особенности оптимизации запросов в распределенных реляционных СУБД

В самой общей постановке задачей компонента распределенной СУБД, оптимизирующего выполнение глобального запроса, является генерация множества альтернативных планов выполнения запроса и выбора из этого множества на основе некоторых критериев одного плана, на основе которого и производится выполнение запроса. Как видно, в такой общей постановке задача оптимизации глобальных запросов формулируется точно так же, как и в случае локальной СУБД. Однако, решение этой задачи для распределенных систем с локальной автономностью узлов существенно более сложное, чем для локальных СУБД.

Для определенности мы будем рассматривать подход к обработке глобальных запросов, основанный на их предварительной компиляции. Этот подход используется, например, в System R\* [93] и состоит в том, что фазы порождения выполняемого плана глобального запроса и его реального выполнения разнесены во времени. Это является естественным развитием подхода System R и позволяет, например, заранее откомпилировать программу с включением глобальных запросов на языке SQL, а затем много раз выполнять ее без необходимости каждый раз вырабатывать планы выполнения запросов.

В результате компиляции запроса в System R\*, инициированной в некотором узле сети, на самом деле порождается распределенная программа выполнения этого запроса, причем она и хранится в распределенной форме. В каждом узле сети, локальная база данных которого содержит отношения, затрагиваемые запросом, хранится часть распределенной программы, под управлением которой производятся доступ к локальным данным этого узла и взаимодействия с другими узлами, содержащими части той же распределенной программы. Выполнение запроса начинается с запуска "главной" части распределенной программы, хранящейся в том узле, в котором инициировалась компиляция запроса ("главном" узле). Эта программа путем сетевого взаимодействия вызывает другие части распределенной программы, хранящиеся в "дополинительных" узлах и т.д. Результат выполнения запроса формируется в главном узле, хотя промежуточные результаты могут быть распределены между другими локальными базами данных.

В System R\* распределенная программа - это программа в машинных кодах IBM/370, но в данном случае, в контексте оптимизации запросов, для нас это не важно: программа могла бы порождаться и на некотором промежуточном языке и интерпретироваться при своем выполнении. Такой подход также практически применяется. Примером системы может быть распределенный вариант СУБД INGRES [91]. Задача оптимизации запросов остается той же - необходимо построить оптимальный план выполнения запроса в условиях локальной автономности узлов сети.

Рассмотрим более подробно, как решается эта задача в System R\*. Основой обработки запроса является поддержание распределенного каталога глобальной базы данных. Подробно это описано в [129]. Здесь же заметим лишь то, что за счет наличия правил именования объектов глобальной базы данных и специальных протоколов доступа к локальным каталогам баз данных при обработке запроса можно получить достоверную информацию о всех затрагиваемых запросом объектах базы данных. В принципе после этого можно было бы генерировать полные распределенные планы выполнения запроса и выбирать из них оптимальный в том узле, в котором начата обработка запроса. Но это противоречило бы принципу локальной автономности узлов сети.

Действительно, предположим, что в построенном детальном плане выполнения запроса предполагается сканирование некоторого удаленного отношения R с использованием индекса I. Естественно, что во время выполнения это сканирование должно производиться в локальной СУБД, база данных которой содержит отношение R. В соответствии с требованием локальной автономности локальный администратор этой базы данных может уничтожить индекс I, и тогда для того, чтобы привести выполняемый план в корректное состояние, потребуется взаимодействие с главным узлом, что противоречит требованиям локальной автономности.

В System R\* применяется следующее достаточно естественное решение: При обработке запроса в главном узле генерируются не детальные, а так называемые *глобальные* планы выполнения запросов. Каждый глобальный план соответствует отдельной схеме межузловых взаимодействий при выполнении запроса. В нем определяются узлы, в которых должны выполняться соединения удаленных отношений, и определяются методы и порядок передачи кортежей между узлами. Вместе с тем, глобальный план выполнения запроса не предписывает правил выполнения локальных реляционных операций в узлах, включаемых в выполнение запроса.

Конечно, как и в случае построения детального плана выполнения запроса, порождается множество альтернативных планов, которые необходимо оценивать. Для оценок используется информация распределенного каталога. По существу, необходимо оценить мощности промежуточных отношений, порождаемых в удаленных узлах при выполнении локальных частей запроса. В отличие от случая локальной СУБД, оценочные формулы глобальных планов в основном базируются на оценках сетевых накладных расходов.

После порождения множества альтернативных глобальных планов запроса, вычисления оценок их стоимости и выбора наиболее дешевого завершается первая фаза оптимизации глобального запроса и начинается следующая фаза - построение детального распределенного плана.

Для этого прежде всего производится декомпозиция исходного запроса в соответствии с выбранным глобальным планом на компоненты, каждый из которых содержит некоторый локальный подзапрос исходного запроса в непроцедурной форме и процедурную часть, предписывающую порядок сетевых взаимодействий. Полученные компоненты рассылаются по сети в соответствующие локальные СУБД, в каждой из которых осуществляется генерация альтернативных *локальных* планов выполнения подзапроса и выбор наиболее дешевого из них в соответствии с локальными критериями оценок стоимости.

Эта процедура практически совпадает с той, которая выполняется при обработке локального запроса в обычной System R. Заметим, что поскольку порядок сетевых взаимодействий является заранее предписанным, то это является некоторым граничным условиям выбора алтернативных локальных планов. Из этого следует, в частности, что в оценочных формулах локальных СУБД не требуется учитывать стоимость сетевых накладных расходов - оно одна и та же для всех возможных локальных планов.

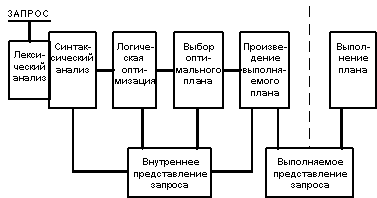
Вторая фаза оптимизации происходит тем самым в распределенной манере. В ней участвуют в общем случае локальная СУБД главного узла и несколько локальных СУБД дополнительных узлов. Если запрос производится не только над базовыми отношениями, а и над представлениями, то раскрытие этого представления производится в том узле, в котором оно определялось, и в общем случае, если это представление определено над несколькими удаленными отношениями, дополнительный узел выступает для своего подзапроса как главный, т.е. вырабатывает глобальный план выполнения подзапроса и рассылает его компоненты дополнительным узлам следующего уровня. Более подробно обработка запросов над представлениями рассматривается в [129].

Заметим, что в этой существенно более сложной, чем в локальном случае, схеме оптимизации не возникают дополнительные проблемы по части оценок стоимости планов выполнения запросов. Основная проблема остается той же, что и при оптимизации в локальной СУБД,- точность оценок селективности простых предикатов. Поэтому и подходы к решению этой проблемы могут быть такими же, как рассмотренные в Разделе 2. Конечно, если использовать для оценок селективности методы, основанные на гистограммах, то при выборе глобального плана выполнения запроса могут потребоваться дополнительные сетевые накладные расходы.

В System R\*, как и в System R, оценки селективность основаны на предположениях о равномерности и независимости распределений значений полей отношений. Эти предположения не вполне правомерны, но зато резко упрощают проблемы оценок. Как отмечается в [95], экспериментальные результаты использования System R\* показывают достаточную достоверность оценок оптимизатора (здесь, конечно, нужно учитывать экспериментальный характер баз данных), но отмечают недостаточное развитие используемых стратегий выполнения соединений удаленных отношений. Подходы к улучшению стратегий мы рассмотрим в следующем разделе.

#### Общая схема обработки запроса

Можно представить, что обработка поступившего в систему запроса состоит из фаз, изображенных ниже.



На первой фазе запрос, заданный на языке запросов, подвергается лексическому и синтаксическому анализу. При этом вырабатывается его внутреннее представление, отражающее структуру запроса и содержащее информацию, которая характеризует объекты базы данных, упомянутые в запросе (отношения, поля и константы). Информация о хранимых в базе данных объектах выбирается из каталогов базы данных. Внутреннее представление запроса используется и преобразуется на следующих стадиях обработки запроса. Форма внутреннего представления должна быть достаточно удобной для последующих оптимизационных преобразований.

На второй фазе запрос во внутреннем представлении подвергается логической оптимизации. Могут применяться различные преобразования, "улучшающие" начальное представление запроса. Среди преобразований могут быть эквивалентные, после проведения которых получается внутреннее представление, семантически эквивалентное начальному (например, приведение запроса к некоторой канонической форме), Преобразования могут быть и семантическими: получаемое представление не является семантически эквивалентным начальному, но гарантируется, что результат выполнения преобразованного запроса совпадает с результатом запроса в начальной форме при соблюдении ограничений целостности, существующих в базе данных. После выполнения второй фазы обработки запроса его внутреннее представление остается непроцедурным, хотя и является в некотором смысле более эффективным, чем начальное.

Третий этап обработки запроса состоит в выборе на основе информации, которой располагает оптимизатор, набора альтернативных процедурных планов выполнения данного запроса в соответствии с его внутреннем представлением, полученным на второй фазе. Для каждого плана оценивается предполагаемая стоимость выполнения запроса. При оценках используется статистическая информация о состоянии базы данных, доступная оптимизатору. Из полученных альтернативных планов выбирается наиболее дешевый, и его внутреннее представление теперь соответствует обрабатываемому запросу.

На четвертом этапе по внутреннему представлению наиболее оптимального плана выполнения запроса формируется выполняемое представление плана. Выполняемое представление плана может быть программой в машинных кодах, если, как в случае System R, система ориентирована на компиляцию запросов в машинные коды, или быть машинно-независимым, но более удобным для интерпретации, если, как в случае INGRES, система ориентирована на интерпретацию запросов. В нашем случае это непринципиально, поскольку четвертая фаза обработки запроса уже не связана с оптимизацией.

Наконец, на пятом этапе обработки запроса происходит его реальное выполнение. Это либо выполнение соответствующей подпрограммы, либо вызов интерпретатора с передачей ему для интерпретации выполняемого плана.

## 22. Архитектура "клиент-сервер". Серверы баз данных.

**Архитектура "клиент-сервер".**

Как правило компьютеры и программы, входящие в состав информационной системы, не являются равноправными. Некоторые из них владеют ресурсами (файловая система, процессор, принтер, база данных и т.д.), другие имеют возможность обращаться к этим ресурсам. Компьютер (или программу), управляющий ресурсом, называют сервером этого ресурса (файл-сервер, сервер базы данных, вычислительный сервер...). Клиент и сервер какого-либо ресурса могут находится как в рамках одной вычислительной системы, так и на различных компьютерах, связанных сетью.

Основной принцип технологии "клиент-сервер" заключается в разделении функций приложения на три группы:

* ввод и отображение данных (взаимодействие с пользователем);
* прикладные функции, характерные для данной предметной области;
* функции управления ресурсами (файловой системой, базой даных и т.д.)

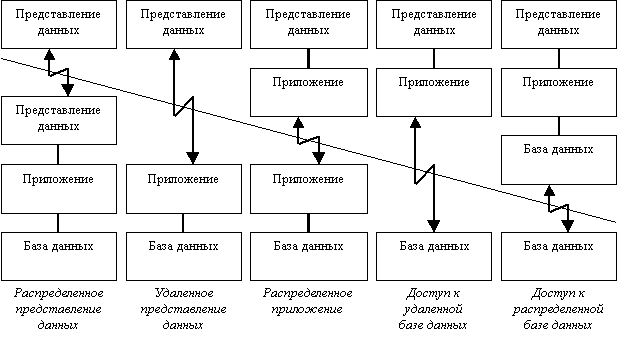
Поэтому, в любом приложении выделяются следующие компоненты:

* компонент представления данных
* прикладной компонент
* компонент управления ресурсом

Связь между компонентами осуществляется по определенным правилам, которые называют "протокол взаимодействия".

**Модели взаимодействия клиент-сервер.**

Компанией [Gartner Group](http://www.gartner.com/), специализирующейся в области исследования информационных технологий, предложена следующая классификация двухзвенных моделей взаимодействия клиент-сервер (двухзвенными эти модели называются потому, что три компонента приложения различным образом распределяются между двумя узлами):



Исторически первой появилась модель распределенного представления данных, которая реализовывалась на универсальной ЭВМ с подключенными к ней неинтеллектуальными терминалами. Управление данными и взаимодействие с пользователем при этом объединялись в одной программе, на терминал передавалась только "картинка", сформированная на центральном компьютере.

Затем, с появлением персональных компьютеров (ПК) и локальных сетей, были реализованы модели доступа к удаленной базе данных. Некоторое время базовой для сетей ПК была архитектура файлового сервера. При этом один из компьютеров является файловым сервером, на клиентах выполняются приложения, в которых совмещены компонент представления и прикладной компонент (СУБД и прикладная программма). Протокол обмена при этом представляет набор низкоуровненых вызовов операций файловой системы. Такая архитектура, реализуемая, как правило, с помощью персональных СУБД, имеет очевидные недостатки - высокий сетевой трафик и отсутствие унифицированного доступа к ресурсам.

С появлением первых специализированных серверов баз данных появилась возможность другой реализации модели доступа к удаленной базе данных. В этом случае ядро СУБД функционирует на сервере, протокол обмена обеспечивается с помощью языка SQL. Такой подход по сравнению с файловым сервером ведет к уменьшению загрузки сети и унификации интерфейса "клиент-сервер". Однако, сетевой трафик остается достаточно высоким, кроме того, по прежнему невозможно удовлетворительное администрирование приложений, поскольку в одной программе совмещаются различные функции.

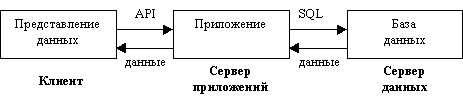
Позже была разработана концепция активного сервера, который использовал механизм хранимых процедур. Это позволило часть прикладного компонента перенести на сервер (модель распределенного приложения). Процедуры хранятся в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, что и SQL-сервер. Преимущества такого подхода: возможно централизованное администрирование прикладных функций, значительно снижается сетевой трафик (т.к. передаются не SQL-запросы, а вызовы хранимых процедур). Недостаток - ограниченность средств разработки хранимых процедур по сравнению с языками общего назначения (C и Pascal).

На практике сейчас обычно используются смешанный подход:

* простейшие прикладные функции выполняются хранимыми процедурами на сервере
* более сложные реализуются на клиенте непосредственно в прикладной программе

Сейчас ряд поставщиков коммерческих СУБД объявило о планах реализации механизмов выполнения хранимых процедур с использованием языка Java. Это соответствует концепции "тонкого клиента", функцией которого остается только отображение данных (модель удаленного представления данных).

В последнее время также наблюдается тенденция ко все большему использованию модели распределенного приложения. Характерной чертой таких приложений является логическое разделение приложения на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате. В этом случае двухзвенная архитектура клиент-сервер становится трехзвенной, а к некоторых случаях, она может включать и больше звеньев.



**Мониторы транзакций.**

В том случае, когда информационная система объединяет достаточно большое количество различных информационных ресурсов и серверов приложений, встает вопрос об оптимальном управлении всеми ее компонентами. В этом случае используют специализированные средства - мониторы обработки транзакций (часто их называют просто "мониторы транзакций"). При этом понятие транзакции расширяется по сравнению с используемым в теории баз данных. В данном случае это не атомарное действие над базой данных, а любое действие в системе - выдача сообщения, запись в индексный файл, печать отчета и т.д.

Для общения прикладной программы с монитором транзакций используется специализированный API (Application Program Interface - интерфейс прикладного программмирования), который реализуется в виде библиотеки, содержащей вызовы основных функций (установить соединение, вызвать определенный сервис и т.д.). Серверы приложений (сервисы) также создаются с помощью этого API, каждому сервису присваивается уникальное имя. Монитор транзакций, получив запрос от прикладной программы, передает ее вызов соответствующему сервису (если тот не запущен, порождается необходимый процесс), после обработки запроса сервером приложений возвращает результаты клиенту. Для взаимодействия мониторов транзакций с серверами баз данных разработан протокол XA. Наличие такого унифицированного интерфейса позволяет использовать в рамках одного приложения несколько различных СУБД.

Использование мониторов транзакций в больших системах дает следующие преимущества:

* Концентрация всех прикладных функций на сервере приложений обеспечивает значительную независимость как от реализации интерфейса с пользователем, так и от конкретного способа управления ресурсами. При этом также обеспечивается централизованное администрирование приложений, поскольку все приложение находится в одном месте, а не "размазано" по сети по клиентским рабочим местам.
* Монитор транзакций в состоянии сам запускать и останавливать серверы приложений. В зависимости от загрузки сети и вычислительных ресурсов он может перенести или скопировать часть серверных процессов на другие узлы. Это обеспечивает достижение баланса загрузки.
* Обеспечивается динамическая конфигурация системы, т.е. без ее остановки может быть добавлен новый сервер ресурсов или сервер приложений.
* Повышается надежность системы, т.к. в случае сбоев сервер приложений может быть перемещен на резервный компьютер.
* Появляется возможность управления распределенными базами данных (подробнее см. следующий параграф).

## 23. Распределенные БД.

*База данных* – интегрированная совокупность данных, с которой работают много пользователей. Изложение всех предыдущих разделов предполагало единую базу данных, размещаемую на одном компьютере. Напомним основные принципы, положенные в основу теории баз данных:

* централизованное хранение данных;
* централизованное обслуживание данных (ввод, корректировка, чтение, контроль целостности).

Заметим, что *базы данных* появились в период господства больших ЭВМ. *База данных* велась на одной ЭВМ, все пользователи работали именно на ЭВМ (возможные режимы работы описаны в ["Различные архитектурные решения, используемые при реализации многопользовательских СУБД. Краткий обзор СУБД"](https://www.intuit.ru/studies/courses/508/364/lecture/8643)). Других вариантов использования вычислительной техники в то время просто не существовало. Если проанализировать работу пользователей с данными в компаниях, организациях, предприятиях в "докомпьютерное" время, то нетрудно заметить, что на отдельных участках пользователи работали со "своими" данными (осуществляли сбор определенных данных, их хранение, обработку, передачу обработанных данных на другие участки или уровни управления).

У такой технологии были существенные недостатки, которые уже отмечались в предыдущих разделах: дублирование некоторых данных, отсутствие возможности сравнительного анализа данных всех участков. Однако у этой технологии были и существенные достоинства: данные вводились и хранились в местах их порождения; с этими данными работал *пользователь*, являющийся специалистом именно по этим данным, что позволяло ему вести эффективный *контроль* правильности данных на всех стадиях обработки; данные находились непосредственно у пользователя, что давало возможность их оперативной обработки. Централизация данных на одной ЭВМ, несомненно, дающая эффективные возможности хранения и обработки данных, не позволяла реализовывать вышеназванные достоинства.

Развитие вычислительных компьютерных сетей обусловило новые возможности в организации и ведении баз данных, позволяющие каждому пользователю иметь на своем компьютере свои данные и работать с ними и в то же время позволяющие работать всем пользователям со всей совокупностью данных как с единой централизованной базой данных. Соответствующая совокупность данных называется распределенной базой данных.

Термин " **распределенная база данных** " достаточно часто встречается в литературе. Однако в разных источниках под этим термином понимаются совершенно разные вещи. Часть авторов понимают под распределенной базой данных то, что имеется удаленный *сервер*, на котором расположены данные, а также клиентские компьютеры, расположенные территориально в другом месте. Такая трактовка нам представляется неправильной. Настоящая **распределенная база данных**располагается на нескольких компьютерах. При этом часть файлов расположена на одном компьютере, часть на другом и т.д. Более того, возможна и даже часто встречается ситуация, когда *информация* на этих компьютерах пересекается, дублируется.

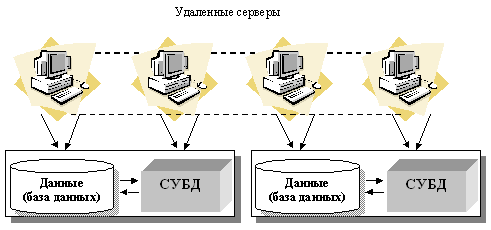
*Распределенная база данных* – совокупность логически взаимосвязанных *разделяемых данных* (и описаний их структур), физически распределенных в компьютерной сети.

*Система управления распределенной базой данных* – программная система, обеспечивающая работу с распределенной базой данных и позволяющая пользователю работать как с его локальными данными, так и со всей базой данных в целом.

*Система управления распределенной базой данных* (РаСУБД) является распределенной системой. Каждый фрагмент *базы данных*работает под управлением отдельной *СУБД*, которая осуществляет *доступ* к данным этого фрагмента. Пользователи взаимодействуют с распределенной базой данных через локальные и глобальные приложения. Локальные приложения дают пользователю возможность работать со своими локальными данными и не требуют доступа к другим фрагментам. Глобальные приложения дают пользователю возможность работать с другими фрагментами *базы данных*, расположенными на других компьютерах сети. Общая схема распределенной *базы данных* представлена на рис. 14.4.

*Объединение* данных организуется виртуально. Соответствующий подход, по сути, отражает организационную структуру предприятия (и даже общества в целом), состоящего из отдельных подразделений. Причем, хотя каждое подразделение обрабатывает свой набор данных (эти наборы, как правило, пересекаются), существует необходимость доступа к этим данным как к единому целому (в частности, для управления всем предприятием).

Одним из примеров реализации такой модели может служить *сеть* *Интернет*: данные вводятся и хранятся на разных компьютерах по всему миру, любой *пользователь* может получить *доступ* к этим данным, не задумываясь о том, где они физически расположены.



**Рис. 14.4.**Распределенная база данных

К.Дж. Дейт провозглашает следующий фундаментальный принцип распределенной *базы данных* . Для пользователя распределенная система должна выглядеть точно так же, как нераспределенная. Из этого принципа следует ряд правил:

1. Локальная автономия.
2. Независимость от центрального узла.
3. Непрерывное функционирование.
4. Независимость от расположения.
5. Независимость от фрагментации.
6. Независимость от репликации.
7. Обработка распределенных запросов.
8. Управление *распределенными транзакциями*.
9. Независимость от аппаратного обеспечения.
10. Независимость от операционной системы.
11. Независимость от сети.
12. Независимость от СУБД.

*Заметим, что понятие распределенной базы данных можно интерпретировать как следующий шаг в развитии понятий о данных* (см. ["Введение в базы данных. Общая характеристика основных понятий"](https://www.intuit.ru/studies/courses/508/364/lecture/8639)), *обусловленный распределенностью данных в реальных предметных областях, а также новым этапом развития средств вычислительной техники – широким использованием вычислительных сетей*.

*В этой интерпретации распределенную базу данных можно понимать как совокупность логически взаимосвязанных распределенных по разным компьютерам баз данных*.

1. Перечислим основные проблемы создания распределенной *базы данных*.
2. Фрагментация данных и распределение по компьютерам.
3. Составление глобального каталога, содержащего информацию о каждом фрагменте БД и его местоположении в сети. (Каталог может храниться на одном узле или быть распределенным)

Организация обработки запросов (синхронизация нескольких запросов к одним и тем же данным, исключение *аномалий удаления* и обновления одних и тех же данных, расположенных на различных узлах, оптимизация последовательности шагов при обработке запроса и т.д.).

Значительным достоинством этой модели является приближение данных к месту их порождения, что позволяет существенно повысить их *достоверность*, недостатком – достаточно высокая сложность управления данными как единым целым.

К сожалению, процесс создания и обслуживания распределенных баз данных связан и с техническими трудностями, среди которых можно выделить жесткие требования к пропускной способности каналов связи, а также низкую *производительность*, обусловленную значительными затратами коммуникационных и вычислительных ресурсов при их синхронизации во *время выполнения* транзакций (особенно при интенсивных обращениях из разных узлов к одному фрагменту).

В задачу данного учебника не входит подробное изучение принципов построения распределенных баз данных. Интересующимся рекомендуем обратиться к соответствующей литературе. Здесь мы хотим лишь обрисовать проблему и сделать некоторые выводы по перспективам ее решения. Технология, связанная с использованием распределенных баз данных, в наибольшей степени соответствует организационной человеческой деятельности (*информация* распределена по месту деятельности людей, и они обмениваются ей в процессе работы) и позволяет наиболее успешно решать важнейшие проблемы ведения баз данных:

* повысить достоверность информации (информация вводится в месте ее порождения лицом, которое лучше всех понимает ее смысловое значение);
* повысить оперативность локальной обработки информации (соответствующие вопросы решаются на локальном компьютере с фрагментом базы данных).

Поэтому очевидно, что задача проектирования, создания и функционирования распределенных баз данных является весьма существенной, активно изучается в настоящее время и будет решаться и далее.

## 24. Системы управления базами данных следующего поколения.

В этом разделе очень кратко рассматриваются основные направления исследований и разработок в области так называемых постреляционных систем, т.е. систем, относящихся к следующему поколению (хотя термин "next-generation DBMS" зарезервирован для некоторого подкласса современных систем).

Хотя отнесение СУБД к тому или иному классу в настоящее время может быть выполнено только условно (например, иногда объектно-ориентированную СУБД O2 относят к системам следующего поколения), можно отметить три направления в области СУБД следующего поколения. Чтобы не изобретать названий, будем обозначать их именами наиболее характерных СУБД.

1. *Направление Postgres*. Основная характеристика: максимальное следование (насколько это возможно с учетом новых требований) известным принципам организации СУБД (если не считать коренной переделки системы управления внешней памятью).
2. *Направление Exodus/Genesis*. Основная характеристика: создание собственно не системы, а генератора систем, наиболее полно соответствующих потребностям приложений. Решение достигается путем создания наборов модулей со стандартизованными интерфейсами, причем идея распространяется вплоть до самых базисовых слоев системы.
3. *Направление Starburst*. Основная характеристика: достижение расширяемости системы и ее приспосабливаемости к нуждам конкретных приложений путем использования стандартного механизма управления правилами. По сути дела, система представляет собой некоторый интерпретатор системы правил и набор модулей-действий, вызываемых в соответствии с этими правилами. Можно изменять наборы правил (существует специальный язык задания правил) или изменять действия, подставляя другие модули с тем же интерфейсом.

В целом можно сказать, что СУБД следующего поколения - это прямые наследники реляционных систем. Тем не менее, различные направления систем третьего поколения стоит рассмотреть отдельно, поскольку они обладают некоторыми разными характеристиками.

#### Ориентация на расширенную реляционную модель

Одним из основных положений реляционной модели данных является требование нормализации отношений: поля кортежей могут содержать лишь атомарные значения. Для традиционных приложений реляционных СУБД - банковских систем, систем резервирования и т.д. - это вовсе не ограничение, а даже преимущество, позволяющее проектировать экономные по памяти БД с предельно понятной структурой. Запросы с соединениями в таких системах сравнительно редки, для динамической поддержки целостности используются соответствующие средства SQL.

Однако с появлением эффективных реляционных СУБД их стали пытаться использовать и в менее традиционных прикладных системах - САПР, системах искусственного интеллекта и т.д. Такие системы обычно оперируют сложно структурированными объектами, для реконструкции которых из плоских таблиц реляционной БД приходится выполнять запросы, почти всегда требующие соединения отношений. В соответствии с требованиями разработчиков нетрадиционных приложений появилось направление исследований баз сложных объектов. Основной смысл этого направления состоит в том, что в руки проектировщиков даются настолько же мощные и гибкие средства структуризации данных, как те, которые были присущи иерархическим и сетевым системам базам данных.

Однако важным отличием является то, что в системах баз данных, поддерживающих сложные объекты, сохраняется четкая граница между логическим и физическим представлениями таких объектов. В частности, для любого сложного объекта (произвольной сложности) должна обеспечиваться возможность перемещения или копирования его как единого целого из одной части базы данных в другую ее часть или даже в другую базу данных. Это очень обширная область исследований, в которой затрагиваются вопросы моделей данных, структур данных, языков запросов, управления транзакциями, журнализации и т.д. Во многом эта область соприкасается с областью объектно-ориентированных БД. (И в этой области настолько же плохо обстоят дела с теоретическим обоснованием.)

Близкое, но, вообще говоря, основанное на других принципах направление представлено системами баз данных, основанных на реляционной модели, в которой не обязательно поддерживается первая нормальная форма отношений. Напомним, что требование атомарности значений, которые могут храниться в элементах кортежей отношений, является базовым требованием классической реляционной модели. Приведение исходного табличного представления предметной области к "плоскому" виду является обязательным первым шагом в процессе проектирования реляционной базы данных на основе принципов нормализации. С другой стороны, абсолютно очевидно, что такое "уплощение" таблиц хотя и является необходимым условием получения неизбыточной и "правильной" схемы реляционной базы данных, в дальнейшем потенциально вызывает выполнение многочисленных соединений, наличие которых может свести на нет все преимущества "хорошей" схемы базы данных.

Так вот, в "ненормализованных" реляционных моделях данных допускается хранение в качестве элемента кортежа кортежей (записей), массивов (регулярных индексированных множеств данных), регулярных множеств элементарных данных, а также отношений. При этом такая вложенность может быть, по существу, неограниченной. Если внимательно продумать эти идеи, то станет понятно, что они приводят (только) к логически обособленным (от физического представления) возможностям иерархической модели данных. Но это уже не так уж и мало, если учесть, что к настоящему времени фактически полностью сформировано теоретическое основание реляционных баз данных с отказом от нормализации. Скорее всего, в этой теории все еще имеются темные места (они наличествуют даже в классической реляционной теории), но тем не менее большинство известных теоретических результатов реляционной теории уже распространено на ненормализованную модель, и даже такой пурист реляционной модели, как Дейт, полагает возможным использование ограниченной и контролируемой реляционной модели в SQL-3.

#### Абстрактные типы данных

Одной из наиболее известных СУБД третьего поколения является система Postgres, а создатель этой системы М.Стоунбрекер, по всей видимости, является вдохновителем всего направления. В Postgres реализованы многие интересные средства: поддерживается темпоральная модель хранения и доступа к данным (см. ниже) и в связи с этим абсолютно пересмотрен механизм журнализации изменений, откатов транзакций и восстановления БД после сбоев; обеспечивается мощный механизм ограничений целостности; поддерживаются ненормализованные отношения (работа в этом направлении началась еще в среде Ingres), хотя и довольно странным способом: в поле отношения может храниться динамически выполняемый запрос к БД.

Одно свойство системы Postgres сближает ее со свойствами объектно-ориентированных СУБД. В Postgres допускается хранение в полях отношений данных абстрактных, определяемых пользователями типов. Это обеспечивает возможность внедрения поведенческого аспекта в БД, т.е. решает ту же задачу, что и ООБД, хотя, конечно, семантические возможности модели данных Postgres существенно слабее, чем у объектно-ориентированных моделей данных. Основная разница состоит в том, что системы класса Postgres не предполагают наличия языка программирования, одинаково понимаемого как внешней системой программирования, так и системой управления базами данных. Если с использованием такой системы программирования определяются типы данных, хранимых в базе данных, то СУБД оказывается не в состоянии контролировать безопасность этих определений, т.е. то отсутствует гарантия, что при выполнении процедур абстрактных типов данных не будет разрушена сама база данных.

Заметим, что в середине 1995 г. компания Sun Microsystems объявила о выпуске нового продукта - языка и семейства интерпретаторов под названием Java. Язык Java является расширенным подмножеством языка Си++. Основные изменения касаются того, что язык является пооператорно интерпретируемым (в стиле языка Бейсик), а программы, написанные на языке Java, гарантированно безопасны (в частности, при выполнении любой программы не может быть поврежден интерпретатор). Для этого, в частности, из языка удалена арифметика над указателями. В то же время Java остается мощным объектно-ориентированным языком, включающим развитые средства определения абстрактных типов данных. Компания Sun продвигает язык Java с целью расширения возможностей службы Всемирной Паутины (World Wide Web) Internet (основная идея состоит в том, что из сервера WWW в клиенты передаются не данные, а объекты, методы которых запрограммированы на языке Java и интерпретируются на стороне клиента; этот подход, в частности, решает проблему нестандартизованного представления мультимедийной информации). Однако, как кажется, интерпретируемый и безопасный язык типа Java может быть успешно применен и в системах баз данных, допускающих хранение данных с типами, определенными пользователями.

#### Генерация систем баз данных, ориентированных на приложения

Идея очень проста: никогда не станет возможно создать универсальную систему управления базами данных, которая будет достаточна и не избыточна для применения в любом приложении. Например, если посмотреть на использование универсальных коммерческих СУБД (например, Oracle или Informix) в российской действительности, то можно легко увидеть, что по крайней мере в 90% случаев применяется не более чем 30% возможностей системы. Тем не менее, приложение несет всю тяжесть поддерживающей его СУБД, рассчитанной на использование в наиболее общих случаях.

Поэтому очень заманчиво производить не законченные универсальные СУБД, а нечто вроде компиляторов компиляторов (сompiler compiler), позволяющих собрать систему баз данных, ориентированную на конкретное приложение (или класс приложений). Рассмотрим простые примеры:

В системах резервирования проездных билетов запросы обычно настолько просты (например, "выдать очередное место на рейс SU 645"), что нет особого смысла производить широкомасштабную оптимизацию запросов. С другой стороны, информация, хранящаяся в базе данных настолько критична (кто из нас не сталкивался с проблемой наличия двух или более билетов на одно место?), что особо важным является гарантированные синхронизация обновлений базы данных и ее восстановление после любого сбоя.

С другой стороны, в статистических системах запросы могут быть произвольно сложными (например, "выдать количество холостых особей мужского пола, проживающих в России и имеющих не менее трех зарегистрированных детей"), что вызывает необходимость использования развитых средств оптимизации запросов. С другой стороны, поскольку речь идет о статистике, здесь не требуется поддержка строгой сериализации транзакций и точного восстановления базы данных после сбоев. (Поскольку речь идет о статистической информации, потеря нескольких ее единиц обычно не существенна.)

Поэтому желательно уметь генерировать систему баз данных, возможности (и соответствующие накладные расходы) которой в достаточной степени соответствуют потребностям приложения. На сегодняшний день на коммерческом рынке такие "генерационные" системы отсутствуют (например, при выборе сервера системы Oracle вы не можете отказаться от каких-либо ненужных для вашего приложения его свойств или потребовать наличия некоторых дополнительных свойств). Однако существуют как минимум два экспериментальных прототипа - Genesis и Exodus.

Обе эти генерационные системы основаны прежде всего на принципах модульности и точного соблюдения установленных интерфейсов. По сути дела, системы состоят из минимального ядра (развитой файловой системы в случае Exodus) и технологического механизма программирования дополнительных модулей. В проекте Exodus этот механизм основывается на системе программирования E, которая является простым расширением Си++, поддерживающим стабильное хранение данных во внешней памяти. Вместо готовой СУБД предоставляется набор "полуфабрикатов" с согласованными интерфейсами, из которых можно сгенерировать систему, максимально отвечающую потребностям приложения.

#### Оптимизация запросов, управляемая правилами

В лекции 18 мы коротко рассмотрели проблемы оптимизации запросов, которые приходится решать в компиляторах языков баз данных. Возможно, главным выводом, который следовало бы сделать на основе материалов этой лекции, является то, что оптимизатор запросов - это наиболее громоздкий, сложный и критичный компонент СУБД. Все разработчики систем управления базами данных согласны с тем, что на оптимизации запросов экономить нельзя. Чем большее количество вариантов выполнения запроса анализируется и чем более точные оценки стоимости плана выполнения запроса применяются, тем более вероятно, что запрос будет выполнен эффективно.

Главная неприятность, связанная с оптимизаторами запросов, состоит в том, что отсутствует принятая технология их программирования. Обычно оптимизатор представляет собой аморфный набор относительно независимых процедур, которые жестко связаны с другими компонентами компилятора. По этой причине очень трудно менять стратегии оптимизации или качественно их расширять (делать это приходится, поскольку оптимизация вообще и оптимизация запросов, в частности, в принципе является эмпирической дисциплиной, а хорошие эмпирические алгоритмы появляются только со временем).

Каким же образом можно решать эту проблему? Имеются компромиссные решения, не выводящие за пределы традиционной технологии производства компиляторов. В основном все они связаны с применением тех или иных инструментальных средств, обеспечивающих автоматизацию построения компиляторов. Среди них отметим технологию, примененную Ричардом Столлманом в его семействе компиляторов gcc, а также инструментальный пакет Cocktail, разработанный в Германском университете города Карлсруе. Основным производственным достоинством gcc является применение единого языка в качестве средства внутреннего представления программы. Высокоуровневый лиспоподобный язык RTL используется на всех фазах компиляции gcc, что позволяет применять одни и те же преобразующие процедуры на разных стадиях оптимизации программы (вплоть до стадии машинно-зависимых оптимизаций).

В пакете Cocktail обеспечивается набор универсальных, настраиваемых процедур преобразования графов внутреннего представления программы. В некотором смысле Cocktail можно рассматривать как специализированный язык для написания компиляторов (компиляторов любых языков, а не только процедурных языков программирования или декларативных языков баз данных). Как утверждается, Cocktail позволяет повысить производительность труда разработчиков компиляторов в 2-3 раза.

Однако наиболее революционный подход среди известных автору был применен в экспериментальной постреляционной системе компании IBM Starburst. В некотором смысле этот подход является развитием идеи Столлмана, примененной при реализации широко популярного редактора Emacs. Напомним, что в основе этого редактора лежит интерпретатор расширенного диалекта языка Common Lisp. Сам этот интерпретатор написан на языке Си, а основная часть редактора написана на языке Лисп. Это позволяет, среди прочего, добавлять в редактор новые возможности, не покидая его среды: вы просто пишете новый текст на Лиспе и объявляете соответствующую функцию подключенной к редактору.

Система Starburst основана на применении продукционной системы. Эта система является, по существу, виртуальной машиной, в которой выполняются все компоненты СУБД, начиная от компилятора языка баз данных (расширенного варианта языка SQL) и заканчивая подсистемой непосредственного исполнения запросов. Сама СУБД представляет собой набор продукционных правил, каждое из которых вызывается продукционной системой при возникновении соответствующего события и выполняет некоторое действие, которое, в свою очередь, может привести к возникновению события, активизирующего другое правило. Правила представляются на специальном языке. Поддерживается набор предопределенных правил низкого уровня, обеспечивающих интерфейс с подсистемой управления внешней памятью (конечно, по соображениям эффективности эта подсистема написана не на продукционном языке).

Очевидно, что такая организация системы обеспечивает максимальную гибкость. Например, чтобы внедрить в оптимизатор запросов некоторую новую стратегию выполнения (например, расширить применяемый набор методов выполнения эквисоединения) достаточно дополнительно написать одно или несколько новых правил, связанных с событием требования выполнить соединение. Тем самым, Starburst может использоваться (и реально используется в научно-исследовательских лабораториях компании IBM) как мощное и гибкое средство исследования методов оптимизации запросов. Конечно, сомнительно, что технология, положенная в основу Starburst, позволит этой системе конкурировать с такими выполненными в традиционной манере коммерческими СУБД, как DB2, Oracle, Informix и т.д.

#### Поддержка исторической информации и темпоральных запросов

Обычные БД хранят мгновенный снимок модели предметной области. Любое изменение в момент времени t некоторого объекта приводит к недоступности состояния этого объекта в предыдущий момент времени. Самое интересное, что на самом деле в большинстве развитых СУБД предыдущее состояние объекта сохраняется в журнале изменений, но возможности доступа со стороны пользователя нет.

Конечно, можно явно ввести в хранимые отношения явный временной атрибут и поддерживать его значения на уровне приложений. Более того, в большинстве случаев так и поступают. Недаром в стандарте SQL появились специальные типы данных date и time. Но в таком подходе имеются несколько недостатков: СУБД не знает семантики временного поля отношения и не может контролировать корректность его значений; появляется дополнительная избыточность хранения (предыдущее состояние объекта данных хранится и в основной БД, и в журнале изменений); языки запросов реляционных СУБД не приспособлены для работы со временем.

Существует отдельное направление исследований и разработок в области темпоральных БД. В этой области исследуются вопросы моделирования данных, языки запросов, организация данных во внешней памяти и т.д. Основной тезис темпоральных систем состоит в том, что для любого объекта данных, созданного в момент времени t1 и уничтоженного в момент времени t2, в БД сохраняются (и доступны пользователям) все его состояния во временном интервале [t1,t2].

Исследования и построения прототипов темпоральных СУБД обычно выполняются на основе некоторой реляционной СУБД. Как и в случае дедуктивных БД темпоральная СУБД - это надстройка над реляционной системой. Конечно, это не лучший способ реализации с точки зрения эффективности, но он прост и позволяет производить достаточно глубокие исследования.

Примером кардинального (но, может быть, преждевременного) решения проблемы темпоральных БД может служить СУБД Postgres. Эта система была спроектирована и разработана М.Стоунбрекером для исследований и обучения студентов в университете г.Беркли, и он безбоязненно шел в ней на самые смелые эксперименты.

Главными особенностями системы управления памятью в Postgres являются, во-первых, то, что в ней не ведется обычная журнализация изменений базы данных и мгновенно обеспечивается корректное состояние базы данных после перевызова системы с утратой состояния оперативной памяти. Во-вторых, система управления памятью поддерживает исторические данные. Запросы могут содержать временные характеристики интересующих объектов. Реализационно эти два аспекта связаны.

Основное решение состоит в том, что при модификациях кортежа изменения производятся не на месте его хранения, а заводится новая запись, куда помещаются измененные поля. Эта запись содержит, кроме того, данные, характеризующие транзакцию, производившую изменения (в том числе и время ее завершения), и подшивается в список к изменявшемуся кортежу. В системе поддерживается уникальная идентификация транзакций и имеется специальная таблица транзакций, хранящаяся в стабильной памяти. Таким образом, после сбоев просто не следует обращать внимание на хвостовые записи списков, относящиеся к незакончившемся транзакциям. Синхронизация поддерживается на основе обычного двухфазного протокола захватов.

Отдельный компонент системы осуществляет архивацию объектов базы данных. Он производит сборку разросшихся списков изменявшихся кортежей и записывает их в область архивного хранения. К этой области тоже могут адресоваться запросы, но уже только на чтение.

Система ориентирована на использование оптических дисков с разовой записью и стабильной оперативной памяти (хотя бы небольшого объема). При наличии таких технических средств она выигрывает по эффективности даже при работе в традиционном режиме по сравнению со схемой с журнализацией. Однако возможна работа и на традиционной аппаратуре, тогда эффективность системы слегка уступает традиционным схемам.

Соответствующие возможности работы с историческими данными заложены в язык Postquel (и в этом его главное отличие от последних вариантов Quel). Возможна выборка информации, хранившейся в базе данных в указанное время, в указанном временном интервале и т.д. Кроме того, имеется возможность создавать версии отношений и допускается их последующая модификация с учетом изменений основных вариантов.

## 25. Объектно-ориентированные СУБД.

Направление объектно-ориентированных баз данных (ООБД) возникло сравнительно давно. Публикации появлялись уже в середине 1980-х. Однако наиболее активно это направление развивается в последние годы. С каждым годом увеличивается число публикаций и реализованных коммерческих и экспериментальных систем.

Возникновение направления ООБД определяется прежде всего потребностями практики: необходимостью разработки сложных информационных прикладных систем, для которых технология предшествующих систем БД не была вполне удовлетворительной.

Конечно, ООБД возникли не на пустом месте. Соответствующий базис обеспечивают как предыдущие работы в области БД, так и давно развивающиеся направления языков программирования с абстрактными типами данных и объектно-ориентированных языков программирования.

Что касается связи с предыдущими работами в области БД, то на наш взгляд наиболее сильное влияние на работы в области ООБД оказывают проработки реляционных СУБД и следующее хронологически за ними семейство БД, в которых поддерживается управление сложными объектами. Кроме того, исключительное влияние на идеи и концепции ООБД и, как кажется, всего объектно-ориентированного подхода оказал подход к семантическому моделированию данных. Достаточное влияние оказывают также развивающиеся параллельно с ООБД направления дедуктивных и активных БД.

Среди языков и систем программирования наибольшее первичное влияние на ООБД оказал Smalltalk. Этот язык сам по себе не является полностью пионерским, хотя в нем была введена новая терминология, являющаяся теперь наиболее распространенной в объектно-ориентированном программировании. На самом деле, Smalltalk основан на ряде ранее выдвинутых концепций.

Большое число опубликованных работ не означает, что все проблемы ООБД полностью решены. Как отмечается в Манифесте группы ведущих ученых, занимающихся ООБД, современная ситуация с ООБД напоминает ситуацию с реляционными системами середины 1970-х. При наличии большого количества экспериментальных проектов (и даже коммерческих систем) отсутствует общепринятая объектно-ориентированная модель данных, и не потому, что нет ни одной разработанной полной модели, а по причине отсутствия общего согласия о принятии какой-либо модели. На самом деле имеются и более конкретные проблемы, связанные с разработкой декларативных языков запросов, выполнением и оптимизацией запросов, формулированием и поддержанием ограничений целостности, синхронизацией доступа и управлением транзакциями и т.д.

Тематика ООБД очень широка, объем этой лекции не позволяет рассмотреть все вопросы. Тем не менее, мы постараемся в систематической манере проанализировать наиболее важные аспекты ООБД.

#### Связь объектно-ориентированных СУБД с общими понятиями объектно-ориентированного подхода

В наиболее общей и классической постановке объектно-ориентированный подход базируется на следующих концепциях:

* объекта и идентификатора объекта;
* атрибутов и методов;
* классов;
* иерархии и наследования классов.

Любая сущность реального мира в объектно-ориентированных языках и системах моделируется в виде объекта. Любой объект при своем создании получает генерируемый системой уникальный идентификатор, который связан с объектом все время его существования и не меняется при изменении состояния объекта.

Каждый объект имеет состояние и поведение. Состояние объекта - набор значений его атрибутов. Поведение объекта - набор методов (программный код), оперирующих над состоянием объекта. Значение атрибута объекта - это тоже некоторый объект или множество объектов. Состояние и поведение объекта инкапсулированы в объекте; взаимодействие объектов производится на основе передачи сообщений и выполнении соответствующих методов.

Множество объектов с одним и тем же набором атрибутов и методов образует класс объектов. Объект должен принадлежать только одному классу (если не учитывать возможности наследования). Допускается наличие примитивных предопределенных классов, объекты-экземпляры которых не имеют атрибутов: целые, строки и т.д. Класс, объекты которого могут служить значениями атрибута объектов другого класса, называется доменом этого атрибута.

Допускается порождение нового класса на основе уже существующего класса - наследование. В этом случае новый класс, называемый подклассом существующего класса (суперкласса), наследует все атрибуты и методы суперкласса. В подклассе, кроме того, могут быть определены дополнительные атрибуты и методы. Различаются случаи простого и множественного наследования. В первом случае подкласс может определяться только на основе одного суперкласса, во втором случае суперклассов может быть несколько. Если в языке или системе поддерживается единичное наследование классов, набор классов образует древовидную иерархию. При поддержании множественного наследования классы связаны в ориентированный граф с корнем, называемый решеткой классов. Объект подкласса считается принадлежащим любому суперклассу этого класса.

Одной из более поздних идей объектно-ориентированного подхода является идея возможного переопределения атрибутов и методов суперкласса в подклассе (перегрузки методов). Эта возможность увеличивает гибкость, но порождает дополнительную проблему: при компиляции объектно-ориентированной программы могут быть неизвестны структура и программный код методов объекта, хотя его класс (в общем случае - суперкласс) известен. Для разрешения этой проблемы применяется так называемый метод позднего связывания, означающий, по сути дела, интерпретационный режим выполнения программы с распознаванием деталей реализации объекта во время выполнения посылки сообщения к нему. Введение некоторых ограничений на способ определения подклассов позволяет добиться эффективной реализации без потребностей в интерпретации.

Как видно, при таком наборе базовых понятий, если не принимать во внимание возможности наследования классов и соответствующие проблемы, объектно-ориентированный подход очень близок к подходу языков программирования с абстрактными (или произвольными) типами данных.

С другой стороны, если абстрагироваться от поведенческого аспекта объектов, объектно-ориентированный подход весьма близок к подходу семантического моделирования данных (даже и по терминологии). Фундаментальные абстракции, лежащие в основе семантических моделей, неявно используются и в объектно-ориентированном подходе. На абстракции агрегации основывается построение сложных объектов, значениями атрибутов которых могут быть другие объекты. Абстракция группирования - основа формирования классов объектов. На абстракциях специализации/обобщения основано построение иерархии или решетки классов.

Видимо, наиболее важным новым качеством ООБД, которого позволяет достичь объектно-ориентированный подход, является поведенческий аспект объектов. В прикладных информационных системах, основывавшихся на БД с традиционной организацией (вплоть до тех, которые базировались на семантических моделях данных), существовал принципиальный разрыв между структурной и поведенческой частями. Структурная часть системы поддерживалась всем аппаратом БД, ее можно было моделировать, верифицировать и т.д., а поведенческая часть создавалась изолированно. В частности, отсутствовали формальный аппарат и системная поддержка совместного моделирования и гарантирования согласованности этих структурной (статической) и поведенческой (динамической) частей. В среде ООБД проектирование, разработка и сопровождение прикладной системы становится процессом, в котором интегрируются структурный и поведенческий аспекты. Конечно, для этого нужны специальные языки, позволяющие определять объекты и создавать на их основе прикладную систему.

Специфика применения объектно-ориентированного подхода для организации и управления БД потребовала уточненного толкования классических концепций и некоторого их расширения. Это определяется потребностями долговременного хранения объектов во внешней памяти, ассоциативного доступа к объектам, обеспечения согласованного состояния ООБД в условиях мультидоступа и тому подобных возможностей, свойственных базам данных. Выделяются три аспекта, отсутствующие в традиционной парадигме, но требующиеся в ООБД.

Первый аспект касается потребности в средствах спецификации знаний при определении класса (ограничений целостности, правил дедукции и т.п.). Второй аспект - потребность в механизме определения разного рода семантических связей между объектами вообще говоря разных классов. Фактически это означает требование полного распространения на ООБД средств семантического моделирования данных. Потребность в использовании абстракции ассоциирования отмечается и в связи с использовании ООБД в сфере автоматизированного проектирования и инженерии. Наконец, третий аспект связан с пересмотром понятия класса. В контексте ООБД оказывается более удобным рассматривать класс как множество объектов данного типа, т.е. одновременно поддерживать понятия и типа и класса объектов.

Как мы отмечали во введении, в сообществе исследователей ООБД и разработчиков систем отсутствует полное согласие, но в большинстве практических работ используется некоторое расширение объектно-ориентированного подхода.

#### Объектно-ориентированные модели данных

Первой формализованной и общепризнанной моделью данных была реляционная модель Кодда. В этой модели, как и во всех следующих, выделялись три аспекта - структурный, целостный и манипуляционный. Структуры данных в реляционной модели основываются на плоских нормализованных отношениях, ограничения целостности выражаются с помощью средств логики первого порядка и, наконец, манипулирование данными осуществляется на основе реляционной алгебры или равносильного ей реляционного исчисления. Как отмечают многие исследователи, своим успехом реляционная модель данных во многом обязана тому, что опиралась на строгий математический аппарат теории множеств, отношений и логики первого порядка. Разработчики любой конкретной реляционной системы считали своим долгом показать соответствие своей конкретной модели данных общей реляционной модели, которая выступала в качестве меры "реляционности" системы.

Основные трудности объектно-ориентированного моделирования данных проистекают из того, что такого развитого математического аппарата, на который могла бы опираться общая объектно-ориентированная модель данных, не существует. В большой степени поэтому до сих пор нет базовой объектно-ориентированной модели. С другой стороны, некоторые авторы утверждают, что общая объектно-ориентированная модель данных в классическом смысле и не может быть определена по причине непригодности классического понятия модели данных к парадигме объектной ориентированности.

Один из наиболее известных теоретиков в области моделей данных Беери предлагает в общих чертах формальную основу ООБД, далеко не полную и не являющуюся моделью данных в традиционном смысле, но позволяющую исследователям и разработчикам систем ООБД по крайней мере говорить на одном языке (если, конечно, предложения Беери будут развиты и получат поддержку). Независимо от дальнейшей судьбы этих предложений мы считаем полезным кратко их пересказать.

Во-первых, следуя практике многих ООБД, предлагается выделить два уровня моделирования объектов: нижний (структурный) и верхний (поведенческий). На структурном уровне поддерживаются сложные объекты, их идентификация и разновидности связи "isa". База данных - это набор элементов данных, связанных отношениями "входит в класс" или "является атрибутом". Таким образом, БД может рассматриваться как ориентированный граф. Важным моментом является поддержание наряду с понятием объекта понятия значения (позже мы увидим, как много на этом построено в одной из успешных объектно-ориентированных СУБД O2).

Важным аспектом является четкое разделение схемы БД и самой БД. В качестве первичных концепций схемного уровня ООБД выступают типы и классы. Отмечается, что во всех системах, использующих только одно понятие (либо тип, либо класс), это понятие неизбежно перегружено: тип предполагает наличие некоторого множества значений, определяемого структурой данных этого типа; класс также предполагает наличие множества объектов, но это множество определяется пользователем. Таким образом, типы и классы играют разную роль, и для строгости и недвусмысленности требуется одновременная поддержка обоих понятий.

Беери не представляет полной формальной модели структурного уровня ООБД, но выражает уверенность, что текущего уровня понимания достаточно, чтобы формализовать такую модель. Что же касается поведенческого уровня, предложен только общий подход к требуемому для этого логическому аппарату (логики первого уровня недостаточно).

Важным, хотя и недостаточно обоснованным предположением Беери является то, что двух традиционных уровней - схемы и данных - для ООБД недостаточно. Для точного определения ООБД требуется уровень мета-схемы, содержимое которой должно определять виды объектов и связей, допустимых на схемном уровне БД. Мета-схема должна играть для ООБД такую же роль, какую играет структурная часть реляционной модели данных для схем реляционных баз данных.

Имеется множество других публикаций, отноcящихся к теме объектно-ориентированных моделей данных, но они либо затрагивают достаточно частные вопросы, либо используют слишком серьезный для этого обзора математический аппарат (например, некоторые авторы определяют объектно-ориентированную модель данных на основе теории категорий).

Для иллюстрации текущего положения дел мы кратко рассмотрим особенности конкретной модели данных, применяемой в объектно-ориентированной СУБД O2 (это, конечно, тоже не модель данных в классическом смысле).

В O2 поддерживаются объекты и значения. Объект - это пара (идентификатор, значение), причем объекты инкапсулированы, т.е. их значения доступны только через методы - процедуры, привязанные к объектам. Значения могут быть атомарными или структурными. Структурные значения строятся из значений или объектов, представленных своими идентификаторами, с помощью конструкторов множеств, кортежей и списков. Элементы структурных значений доступны с помощью предопределенных операций (примитивов).

Возможны два вида организации данных: классы, экземплярами которых являются объекты, инкапсулирующие данные и поведение, и типы, экземплярами которых являются значения. Каждому классу сопоставляется тип, описывающий структуру экземпляров класса. Типы определяются рекурсивно на основе атомарных типов и ранее определенных типов и классов с применением конструкторов. Поведенческая сторона класса определяется набором методов.

Объекты и значения могут быть именованными. С именованием объекта или значения связана долговременность его хранения (persistency): любые именованные объекты или значения долговременны; любые объект или значение, входящие как часть в другой именованный объект или значение, долговременны.

С помощью специального указания, задаваемого при определении класса, можно добиться долговременности хранения любого объекта этого класса. В этом случае система автоматически порождает значение-множество, имя которого совпадает с именем класса. В этом множестве гарантированно содержатся все объекты данного класса.

Метод - программный код, привязанный к конкретному классу и применимый к объектам этого класса. Определение метода в O2 производится в два этапа. Сначала объявляется сигнатура метода, т.е. его имя, класс, типы или классы аргументов и тип или класс результата. Методы могут быть публичными (доступными из объектов других классов) или приватными (доступными только внутри данного класса). На втором этапе определяется реализация класса на одном из языков программирования O2 (подробнее языки обсуждаются в следующем разделе нашего обзора).

В модели O2 поддерживается множественное наследование классов на основе отношения супертип/подтип. В подклассе допускается добавление и/или переопределение атрибутов и методов. Возможные при множественном наследовании двусмысленности (по именованию атрибутов и методов) разрешаются либо путем переименования, либо путем явного указания источника наследования. Объект подкласса является объектом каждого суперкласса, на основе которого порожден данный подкласс.

Поддерживается предопределенный класс "Оbject", являющийся корнем решетки классов; любой другой класс является неявным наследником класса "Object" и наследует предопределенные методы ("is\_same", "is\_value\_equal" и т.д.).

Специфической особенностью модели O2 является возможность объявления дополнительных "исключительных" атрибутов и методов для именованных объектов. Это означает, что конкретный именованный объект-представитель класса может обладать типом, являющимся подтипом типа класса. Конечно, с такими атрибутами не работают стандартные методы класса, но специально для именованного объекта могут быть определены дополнительные (или переопределены стандартные) методы, для которых дополнительные атрибуты уже доступны. Подчеркивается, что дополнительные атрибуты и методы привязываются не к конкретному объекту, а к имени, за которым в разные моменты времени могут стоять вообще говоря разные объекты. Для реализации исключительных атрибутов и методов требуется развитие техники позднего связывания.

В следующем разделе мы среди прочего рассмотрим особенности языков программирования и запросов системы O2, которые, конечно, тесно связаны со спецификой модели данных.

#### Языки программирования объектно-ориентированных баз данных

Как отмечают многие исследователи и разработчики, объектно-ориентированная система БД представляет собой объединение системы программирования и СУБД (альтернативная, но не более проясняющая суть дела точка зрения состоит в том, что объектно-ориентированная СУБД - это СУБД, основанная на объектно-ориентированной модели данных).

##### Потеря соответствия между языками программирования и языками запросов в реляционных СУБД

Мы уже говорили, что основная практическая надобность в ООБД связана с потребностью в некоторой интегрированной среде построения сложных информационных систем. В этой среде должны отсутствовать противоречия между структурной и поведенческой частями проекта и должно поддерживаться эффективное управление сложными структурами данных во внешней памяти. В отличие от случая реляционных систем, где при создании приложения приходится одновременно использовать ориентированный на работу со скалярными значениями процедурный язык программирования и ориентированный на работу со множествами декларативный язык запросов (это принято называть потерей соответствия - impedance mismatch), языковая среда ООБД - это объектно-ориентированная система программирования, естественно включающая средства работы с долговременными объектами. "Естественность" включения средств работы с БД в язык программирования означает, что работа с долговременными (хранимыми во внешней БД) объектами должна происходить на основе тех же синтаксических конструкций (и с той же семантикой), что и работа со временными, существующими только во время работы программы объектами.

Эта сторона ООБД наиболее близка родственному направлению языков программирования баз данных. Языки программирования ООБД и БД во многих своих чертах различаются только терминологически; существенным отличием является лишь поддержание в языках первого класса подхода к наследованию классов. Кроме того, языки второго класса, как правило, более развиты как в отношении системы типов, так и в отношении управляющих конструкций.

Другим аспектом языкового окружения ООБД является потребность в языках запросов, которые можно было бы использовать в интерактивном режиме. Если доступ к объектам внешней БД в языках программирования ООБД носит в основном навигационный характер, то для языков запросов более удобен декларативный стиль. Декларативные языки запросов к ООБД менее развиты, чем языки программирования ООБД, и при их реализации возникают существенные проблемы. В следующем разделе мы рассмотрим имеющиеся подходы и их ограничения более подробно. Но начнем с языков программирования ООБД.

##### Языки программирования ООБД как объектно-ориентированные языки с поддержкой стабильных (persistent) объектов

К настоящему моменту нам неизвестен какой-либо язык программирования ООБД, который был бы спроектирован целиком заново, начиная с нуля. Естественным подходом к построению такого языка было использование (с необходимыми расширениями) некоторого существующего объектно-ориентированного языка. Начало расцвета направления ООБД совпало с пиком популярности языка Smalltalk-80. Этот язык оказал большое влияние на разработку первых систем ООБД, и, в частности, использовался в качестве языка программирования. Во многом опирается на Smalltalk и известная коммерчески доступная система GemStone.

Трудности с эффективной практической реализацией языка Smalltalk побудили разработчиков систем ООБД к поиску альтернативных базовых языков. Известная близость объектно-ориентированного и функционального подходов к программированию позволяет достаточно успешно опираться на функциональные языки программирования. В частности, язык Лисп (Common Lisp) является основой проекта ORION. В этом проекте Лисп является и инструментальным языком, и базой объектно-ориентированного языка программирования в среде ORION.

Потребности в еще более эффективной реализации заставляют использовать в качестве основы объектно-ориентированного языка языки более низкого уровня. Например, в системе VBASE наряду со специально разработанным языком TDL, предназначенным для определения типов, используется объектно-ориентированное расширение языка Си - COP (C Object Processor). В уже упоминавшемся проекте O2 наряду с функциональным объектно-ориентированным языком программирования используются два объектно-ориентированных расширения языков Бейсик и Си. При этом, насколько можно судить по публикациям, наибольшее распространение среди пользователей этой системы (она уже коммерчески доступна) получил язык CO2, являющийся расширением языка Си. Возможно это связано лишь с широкой (и все более возрастающей) популярностью языка Си (и его объектно-ориентированного потомка Си++), ставшего поистине девизом "настоящих программистов". Может быть причины более глубинны (например, языки более высокого уровня слишком ограничительны для программистов-профессионалов; недаром большинство современных реализаций языков более высокого уровня выполняются именно на языке Си). Тем не менее, современная ситуация именно такова, и мы считаем полезным привести краткое описание основных особенностей языка CO2.

##### Примеры языков программирования ООБД

Прежде всего, CO2 не является полностью самостоятельным языком. Этот язык входит в многоязыковую среду O2 и предназначен для программирования методов ранее определенных классов. Определение классов, сигнатур методов (фактически, прототипов функций в терминологии языка Си) и имен постоянно хранимых значений и объектов производится с использованием отдельного языка определения схемы БД.

Имя любого объекта трактуется как указатель на значение этого объекта; разименование производится с помощью обычного оператора Си '\*'. Доступ к значению объекта возможен только из метода его класса, если только при перечислении методов оператор '\*' не объявлен явно публичным.

Поддерживается операция порождения нового объекта указанного класса. В отличие от языка Си++ в CO2 невозможно совместить создание нового объекта с его инициализаций (понятие метода-конструктора начального значения объекта в CO2 не поддерживается). Для инициализации необходимо либо явно обратиться к соответствующему методу класса с указанием вновь созданного объекта (поддерживается соответствующий механизм "передачи сообщений", означающий на самом деле вызов функции), либо воспользоваться оператором '\*' и явно присвоить новое значение, если '\*' - публичный оператор для данного класса.

CO2 включает средства конструирования значений-кортежей, множеств и списков. Понятие значения-кортежа фактически эквивалентно понятию значения-структуры обычного языка Си (с тем отличием, что элементами кортежа могут являться объекты, множества и списки). Для значений-множеств и списков поддерживаются операции добавления и изъятия элементов, а также набор теоретико-множественных операций (и конкатенации для списков).

Основой манипулирования объектами, хранимыми в БД, является расширенное по сравнению с языком Си средство итерации. Итератор применим к значениям-множествам или спискам. Фактически он означает последовательное применение оператора-тела цикла ко всем элементам множества или списка. Если мы вспомним, что долговременно хранимому классу объектов неявно соответствуют одноименное значение-множество с элементами-объектами данного класса, то становится понятно, что итератор языка CO2 обеспечивает явную навигацию в классах объектов. Единственное, что остается от привычных пользователям СУБД языков запросов, - это ограниченная возможность указания характеристик требуемых в цикле объектов (это делается путем использования оператора разименования и явного указания условий на атрибуты; конечно, для этого нужно, чтобы оператор '\*' был объявлен публичным в данном классе).

Разработчики O2 подчеркивают, что они умышленно сделали CO2 более бедным по возможностям, чем, например, язык Си++, потому что многое по части управления объектами берет на себя общий менеджер объектов системы, явно вызываемый из рабочей программы.

#### Языки запросов объектно-ориентированных баз данных

Потребность в поддержании в объектно-ориентированной СУБД не только языка (или семейства языков) программирования ООБД, но и развитого языка запросов в настоящее время осознается практически всеми разработчиками. Система должна поддерживать легко осваиваемый интерфейс, прямо доступный конечному пользователю в интерактивном режиме.

##### Явная навигация как следствие преодоления потери соответствия

Наиболее распространенный подход к организации интерактивных интерфейсов с объектно-ориентированными системами баз данных основывается на использовании обходчиков. В этом случае конечный интерфейс обычно является графическим. На экране отображается схема (или подсхема) ООБД, и пользователь осуществляет доступ к объектам в навигационном стиле. Некоторые исследователи считают, что в этом случае разумно игнорировать принцип инкапсуляции объектов и предъявлять пользователю внутренность объектов. В большинстве существующих систем ООБД подобный интерфейс существует, но всем понятно, что навигационный язык запросов - это в некотором смысле шаг назад по сравнению с языками запросов даже реляционных систем. Ведутся активные поиски подходов к организации декларативных языков запросов к ООБД.

##### Ненавигационные языки запросов

Беери отмечает существование трех подходов. Первый подход - языки, являющиеся объектно-ориентированными расширениями языков запросов реляционных систем. Наиболее распространены языки с синтаксисом, близким к известному языку SQL. Это связано, конечно, с общим признанием и чрезвычайно широким распространением этого языка. В частности, в своем Манифесте третьего поколения СУБД М. Стоунбрекер и его коллеги по комитету перспективных систем БД утверждают необходимость поддержания SQL-подобного интерфейса во всех СУБД следующего поколения. Мы уже видели, какое влияние оказывает эта точка зрения на развитие языка SQL.

Второй подход основывается на построении полного логического объектно-ориентированного исчисления. По поводу построения такого исчисления имеются теоретические работы, но законченный и практически реализованный язык запросов нам неизвестен. Видимо к этому же направлению строго теоретически обоснованных языков запросов можно отнести и работы, основанные на алгебраической теории категорий.

Наконец, третий подход основывается на применении дедуктивного подхода. В основном это отражает стремление разработчиков к сближению направлений дедуктивных и объектно-ориентированных БД.

Независимо от применяемого для разработки языка запросов подхода перед разработчиками встает одна концептуальная проблема, решение которой не укладывается в традиционное русло объектно-ориентированного подхода. Понятно, что основой для формулирования запроса должен служить класс, представляющий в ООБД множество однотипных объектов. Но что может представлять собой результат запроса? Набор основных понятий объектно-ориентированного подхода не содержит подходящего к данному случаю понятия. Обычно из положения выходят, расширяя базовый набор концепций множества объектов и полагая, что результатом запроса является некоторое подмножество объектов-экземпляров класса. Это довольно ограничительный подход, поскольку автоматически исключает возможность наличия в языке запросов средств, аналогичных реляционному оператору соединения. Кратко рассмотрим особенности нескольких конкретных декларативных языков запросов к ООБД.

В языке запросов объектно-ориентированной СУБД ORION полностью поддерживается принцип инкапсуляции объектов. В реализованном варианте языка запросы могут основываться только на одном классе (предлагался подход к определению запроса на нескольких классах в стиле расширения семантики реляционного оператора соединения). Синтаксис языка ориентирован на SQL. Очень развит набор допустимых предикатов селекции. В частности, для атрибута, доменом которого является суперкласс, можно указать имя интересующего пользователя подкласса.

Язык запросов системы Iris находится в значительной степени под влиянием реляционной парадигмы. Даже название этого языка OSQL отражает его тесную связь с реляционным языком SQL. По сути дела, OSQL - это реляционный язык, рассчитанный на работу с ненормализованными отношениями. Естественно, при таком подходе в OSQL нарушается инкапсуляция объектов.

На наш взгляд, особый интерес представляет декларативный язык запросов системы O2 RELOOP. В общих словах, это декларативный язык запросов с SQL-ори-ентированным синтаксисом, основанный на специально разработанной для модели O2 алгебре объектов и значений. (Кстати, это не единственная работа в направлении построения алгебры для объектно-ориентированных моделей данных.) Особенно впечатляющим качеством языка RELOOP является естественность его построения в общем контексте модели O2. Запрос задается всегда на значении-множестве или списке. Если мы вспомним, что долговременному классу в O2 соответствует одноименное значение-множество, то тем самым можно определить запрос на любом хранимом классе. Результатом запроса может являться объект, значение-множество или значение-список. При этом элементами значений-множеств могут являться объекты (простая выборка), либо значения-кортежи с элементами-объектами разных классов (например). В совокупности эти особенности языка позволяют формулировать запросы над несколькими классами (специфическое соединение, порождающее не новые объекты, а кортежи из существующих объектов), а также употреблять вложенные подзапросы.

##### Проблемы оптимизации запросов

Как обычно, основной целью оптимизации запроса в системе ООБД является создание оптимального плана выполнения запроса с использованием примитивов доступа к внешней памяти ООБД.

Оптимизация запросов хорошо исследована и разработана в контексте реляционных БД. Известны методы синтаксической и семантической оптимизации на уровне непроцедурного представления запроса, алгоритмы выполнения элементарных реляционных операций, методы оценок стоимости планов запросов.

Конечно, объекты могут иметь существенно более сложную структуру, чем кортежи плоских отношений, но не это различие является наиболее важным. Основная сложность оптимизации запросов к ООБД следует из того, что в этом случае условия выборки формулируются в терминах "внешних" атрибутов объектов (методов), а для реальной оптимизации (т.е. для выработки оптимального плана) требуются условия, определенные на "внутренних" атрибутах (переменных состояния).

На самом деле похожая ситуация существует и в РСУБД при оптимизации запроса над представлением БД. В этом случае условия также формулируются в терминах внешних атрибутов (атрибутов представления), и в целях оптимизации запроса эти условия должны быть преобразованы в условия, определенные на атрибутах хранимых отношений. Хорошо известным методом такой "предоптимизации" является подстановка представлений, которая часто (хотя и не всегда в случае использования языка SQL) обеспечивает требуемые преобразования. Альтернативным способом выполнения запроса над представлением (иногда единственным возможным) является материализация представления.

В системах ООБД ситуация существенно усложняется двумя обстоятельствами. Во-первых, методы обычно программируются на некотором процедурном языке программирования и могут иметь параметры. Т.е. в общем случае тело метода представляет из себя не просто арифметическое выражение, как в случае определения атрибутов представления, а параметризованную программу, включающую ветвления, вызовы функций и методов других объектов. Вторая сложность связана с возможным и распространенным в ООП поздним связыванием: точная реализация метода и даже структура объекта может быть неизвестна во время компиляции запроса.

Одним из подходов к упрощению проблемы является открытие видимости некоторых (наиболее важных для оптимизации) внутренних атрибутов объектов. В этом контексте достаточно было бы открыть видимость только для компилятора запросов, т.е. фактически запретить переопределять такие переменные в подклассах. С точки зрения пользователя такие атрибуты выглядели бы как методы без параметров, возвращающие значение соответствующего типа. С нашей точки зрения лучше было бы сохранить строгую инкапсуляцию объектов (чтобы избавить приложение от критической зависимости от реализации) и обеспечить возможности тщательного проектирования схемы ООБД с учетом потребностей оптимизации запросов.

Общий подход к предоптимизации условия выборки для одного (супер)класса объектов может быть следующим (мы предполагаем, что условия формулируются с использованием логики предикатов первого порядка без кванторов; в предикатах могут использоваться методы соответствующего класса, константы и операции сравнения):

**Шаг А:** Преобразовать логическую формулу условия к конъюнктивной нормальной форме (КНФ). Мы не останавливаемся на способе выбора конкретной КНФ, но естественно, должна быть выбрана "хорошая" КНФ (например, содержащая максимальное число атомарных конъюнктов).

**Шаг B:** Для каждого конъюнкта, включающего методы с известным во время компиляции телом, заменить вызовы методов на их тела с подставленными параметрами. (Для простоты будем предполагать, что параметры не содержат вызовов функций или методов других объектов.)

**Шаг C:** Для каждого такого конъюнкта произвести все возможные упрощения, т.е. вычислить все, что можно вычислить в статике. Хотя в общем виде эта задача является очень сложной, при разумном проектировании ООБД в число методов должны будут войти методы с предельно простой реализацией, задавать условия на которых будет очень естественно. Такие условия будут упрощаться очень эффективно.

**Шаг D:** Если теперь появились конъюнкты, представляющие собой простые предикаты сравнения на основе переменных состояния и констант, использовать эти конъюнкты для выработки оптимального плана выполнения запроса. Если же такие конъюнкты получить не удалось, единственным способом "отфильтровать" (супер)класс объектов является его последовательный просмотр с полным вычислением (возможно упрощенного) логического выражения для каждого объекта.

Понятно, что возможности оптимизации будут зависеть от особенностей языка программирования, который используется для программирования методов, от особенностей конкретного языка запросов и от того, насколько продуманно спроектирована схема ООБД. В частности, желательно, чтобы используемый язык программирования стимулировал максимально дисциплинированный стиль программирования методов объектов. Язык запросов должен разумно ограничивать возможности пользователей (в частности, в отношении параметров методов, участвующих в условиях запросов). Наконец, в классах схемы ООБД должны содержаться простые методы, не переопределяемые в подклассах и основанные на тех переменных состояния, которые служат основой для организации методов доступа.

Заметим, что указанные ограничения не влекут зависимости прикладной программы от особенностей реализации ООБД, поскольку объекты остаются полностью инкапсулированными. Использование в условиях запросов простых методов должно стимулироваться не требованиями реализации, а семантикой объектов.

#### Примеры объектно-ориентированных СУБД

В настоящее время ведется очень много экспериментальных и производственных работ в области объектно-ориентированных СУБД. Больше всего университетских работ, которые в основном носят исследовательский характер. Но уже несколько лет назад отмечалось существование по меньшей мере тринадцати коммерчески доступных систем ООБД. Среди них уже упоминавшиеся в нашем обзоре системы O2, ORION, GemStone и Iris.

Рассмотрим особенности организации двух из них - ORION и O2.

##### Проект ORION

Проект ORION осуществлялся с 1985 по 1989 г. фирмой MCC под руководством известного еще по работам в проекте System R Вона Кима. Под названием ORION на самом деле скрывается семейство трех СУБД: ORION-1 - однопользовательская система; ORION-1SX, предназначенная для использования в качестве сервера в локальной сети рабочих станций; ORION-2 - полностью распределенная объектно-ориентированная СУБД. Реализация всех систем производилась с использованием языка Common Lisp на рабочих станциях (и их локальных сетях) Symbolics 3600 с ОС Genera 7.0 и SUN-3 в среде ОС UNIX.

Основными функциональными компонентами системы являются подсистемы управления памятью, объектами и транзакциями. В ORION-1 все компоненты, естественно, располагаются на одной рабочей станции; в ORION-1SX - разнесены между разными рабочими станциями (в частности, управление объектами производится на рабочей станции-клиенте). Применение в ORION-1SX для взаимодействия клиент-сервер механизма удаленного вызова процедур позволило использовать в этой системе практически без переделки многие модули ORION-1. Сетевые взаимодействия основывались на стандартных средствах операционных систем.

В число функций подсистемы управления памятью входит распределение внешней памяти, перемещение страниц из буферов оперативной памяти во внешнюю память и наоборот, поиск и размещение объектов в буферах оперативной памяти (как принято в объектно-ориентированных системах, поддерживаются два представления объектов - дисковое и в оперативной памяти; при перемещении объекта из буфера страниц в буфер объектов и обратно представление объекта изменяется). Кроме того, эта подсистема ответственна за поддержание вспомогательных индексных структур, предназначенных для ускорения выполнения запросов.

Подсистема управления объектами включает подкомпоненты обработки запросов, управления схемой и версиями объектов. Версии поддерживаются только для объектов, при создании которых такая необходимость была явно указана. Для схемы БД версии не поддерживаются; при изменении схемы отслеживается влияние этого изменения на другие компоненты схемы и на существующие объекты. При обработке запросов используется техника оптимизации, аналогичная применяемой в реляционных системах (т.е. формируется набор возможных планов выполнения запроса, оценивается стоимость каждого из них и выбирается для выполнения наиболее дешевый).

Подсистема управления транзакциями обеспечивает традиционную сериализуемость транзакций, а также поддерживает средства журнализации изменений и восстановления БД после сбоев. Для сериализации транзакций применяется разновидность двухфазного протокола синхронизационных захватов с различной степенью гранулированности. Конечно, при синхронизации учитывается специфика ООБД, в частности, наличие иерархии классов. Журнал изменений обеспечивает откаты индивидуальных транзакций и восстановление БД после мягких сбоев (архивные копии БД для восстановления после поломки дисков не поддерживаются).

##### Проект O2

Проект O2 выполнялся французской компанией Altair, образованной специально для целей проектирования и реализации объектно-ориентированной СУБД. Начало проекта датируется сентябрем 1986 г., и он был рассчитан на пять лет: три года на прототипирование и два года на разработку промышленного образца. После успешного завершения проекта для сопровождения системы и ее дальнейшего развития была организована новая чисто коммерческая компания O2.

Прототип системы функционировал в режиме клиент/сервер в локальной сети рабочих станций SUN c соответствующим разделением функций между сервером и клиентами.

Основными компонентами системы (не считая развитого набора интерфейсных средств) являются интерпретатор запросов и подсистемы управления схемой, объектами и дисками. Управление дисками, т.е. поддержание базовой среды постоянного хранения обеспечивает система WiSS, которую разработчики O2 перенесли в окружение ОС UNIX.

Наибольшую функциональную нагрузку несет компонент управления объектами. В число функций этой подсистемы входят:

* управление сложными объектами, включая создание и уничтожение объектов, выборку объектов по именам, поддержку предопределенных методов, поддержку объектов со внутренней структурой-множеством, списком и кортежем;
* управление передачей сообщений между объектами;
* управление транзакциями;
* управление коммуникационной средой (на базе транспортных протоколов TCP/IP в локальной сети Ethernet);
* отслеживание долговременно хранимых объектов (напомним, что в O2 объект хранится во внешней памяти до тех пор, пока достижим из какого-либо долговременно хранимого объекта);
* управление буферами оперативной памяти (аналогично ORION, представление объекта в оперативной памяти отличается от его представления на диске);
* управление кластеризацией объектов во внешней памяти;
* управление индексами.

Несколько слов про управление транзакциями. Различаются режимы, когда допускается параллельное выполнение транзакций, изменяющих схему БД, и когда параллельно выполняются только транзакции, изменяющие внутренность БД. Первый режим обычно используется на стадии разработки БД, второй - на стадии выполнения приложений. Средства восстановления БД после сбоев и откатов транзакций также могут включаться и выключаться. Наконец, поддерживается режим, при котором все постоянно хранимые объекты загружаются в оперативную память при начале транзакции для увеличения скорости работы прикладной системы.

Компонент управления схемой БД реализован над подсистемой управления объектами: в системе поддерживаются несколько невидимых для программистов классов и в том числе классы "Class" и "Method", экземплярами которых являются, соответственно, объекты, определяющие классы, и объекты, определяющие методы. (Как видно, ситуация напоминает реляционные системы, в которых тоже обычно поддерживаются служебные отношения-каталоги, описывающие схему БД.) Удаление класса, который не является листом иерархии классов или используется в другом классе или сигнатуре какого-либо метода, запрещено.

Даже приведенное краткое описание особенностей двух объектно-ориентированных СУБД показывает прагматичность современного подхода к организации таких систем. Их разработчики не стремятся к полному соблюдению чистоты объектно-ориентированного подхода и применяют наиболее простые решения проблем. Пока в сообществе разработчиков объектно-ориентированных систем БД не видно работы, которая могла бы сыграть в этом направлении роль, аналогичную роли System R по отношению к реляционным системам. Правда и проблемы ООБД гораздо более сложны, чем решаемые в реляционных системах.

## 26. Системы баз данных, основанные на правилах. Активные и дедуктивные базы данных.

В этой очень краткой лекции мы рассмотрим последнюю тему этого курса - системы баз данных, основанные на правилах. Более точно можно было бы сказать, что наша завершающая лекция посвящается системам баз данных, в которых правила играют существенно более важную роль, чем в традиционных реляционных системах. Это уточнение необходимо по той причине, что правила используются для разных целей в любой развитой СУБД.

#### Экстенсиональная и интенсиональная части базы данных

Если внимательно присмотреться к тому, что реально хранится в базе данных, то можно заметить наличие трех различных видов информации. Во-первых, это информация, характеризующая структуры пользовательских данных (описание структурной части схемы базы данных). Такая информация в случае реляционной базы данных сохраняется в системных отношениях-каталогах и содержит главным образом имена базовых отношений и имена и типы данных их атрибутов. Во-вторых, это собственно наборы кортежей пользовательских данных, сохраняемых в определенных пользователями отношениях. Наконец, в-третьих, это правила, определяющие ограничения целостности базы данных, триггеры базы данных и представляемые (виртуальные) отношения. В реляционных системах правила опять же сохраняются в системных таблицах-каталогах, хотя плоские таблицы далеко не идеально подходят для этой цели.

Информация первого и второго вида в совокупности явно описывает объекты (сущности) реального мира, моделируемые в базе данных. Другими словами, это явные факты, предоставленные пользователями для хранения в БД. Эту часть базы данных принято называть экстенсиональной.

Информация третьего вида служит для руководства СУБД при выполнении различного рода операций, задаваемых пользователями. Ограничения целостности могут блокировать выполнение операций обновления базы данных, триггеры вызывают автоматическое выполнение специфицированных действий при возникновении специфицированных условий, определения представлений вызывают явную или косвенную материализацию представляемых таблиц при их использовании. Эту часть базы данных принято называть интенсиональной; она содержит не непосредственные факты, а информацию, характеризующую семантику предметной области.

Как видно, в реляционных базах данных наиболее важное значение имеет экстенсиональная часть, а интенсиональная часть играет в основном вспомогательную роль. В системах баз данных, основанных на правилах, эти две части как минимум равноправны.

#### Активные базы данных

По определению БД называется активной, если СУБД по отношению к ней выполняет не только те действия, которые явно указывает пользователь, но и дополнительные действия в соответствии с правилами, заложенными в саму БД.

Легко видеть, что основа этой идеи содержалась в языке SQL времени System R. На самом деле, что есть определение триггера или условного воздействия, как не введение в БД правила, в соответствии с которым СУБД должна производить дополнительные действия? Плохо лишь то, что на самом деле триггеры не были полностью реализованы ни в одной из известных систем, даже и в System R. И это не случайно, потому что реализация такого аппарата в СУБД очень сложна, накладна и не полностью понятна.

Среди вопросов, ответы на которые до сих пор не получены, следующие. Как эффективно определить набор вспомогательных действий, вызываемых прямым действием пользователя? Каким образом распознавать циклы в цепочке "действие-условие-действие-..." и что делать при возникновении таких циклов? В рамках какой транзакции выполнять дополнительные условные действия и к бюджету какого пользователя относить возникающие накладные расходы?

Масса проблем не решена даже для сравнительно простого случая реализации триггеров SQL, а задача ставится уже гораздо шире. По существу, предлагается иметь в составе СУБД продукционную систему общего вида, условия и действия которой не ограничиваются содержимым БД или прямыми действиями над ней со стороны пользователя. Например, в условие может входить время суток, а действие может быть внешним, например, вывод информации на экран оператора. Практически все современные работы по активным БД связаны с проблемой эффективной реализации такой продукционной системы.

Вместе с тем, по нашему мнению, гораздо важнее в практических целях реализовать в реляционных СУБД аппарат триггеров. Заметим, что в проекте стандарта SQL3 предусматривается существование языковых средств определения условных воздействий. Их реализация и будет первым практическим шагом к активным БД (уже появились соответствующие коммерческие реализации).

#### Дедуктивные базы данных

По определению, дедуктивная БД состоит из двух частей: экстенциональной, содержащей факты, и интенциональной, содержащей правила для логического вывода новых фактов на основе экстенциональной части и запроса пользователя.

Легко видеть, что при таком общем определении SQL-ориентированную реляционную СУБД можно отнести к дедуктивным системам. Действительно, что есть определенные в схеме реляционной БД представления, как не интенциональная часть БД. В конце концов не так уж важно, какой конкретный механизм используется для вывода новых фактов на основе существующих. В случае SQL основным элементом определения представления является оператор выборки языка SQL, что вполне естественно, поскольку результатом оператора выборки является порождаемая таблица. Обеспечивается и необходимая расширяемость, поскольку представления могут определяться не только над базовыми таблицами, но и над представлениями.

Основным отличием реальной дедуктивной СУБД от реляционной является то, что и правила интенциональной части БД, и запросы пользователей могут содержать рекурсию. Можно спорить о том, всегда ли хороша рекурсия. Однако возможность определения рекурсивных правил и запросов дает возможность простого решения в дедуктивных базах данных проблем, которые вызывают большие проблемы в реляционных системах (например, проблемы разборки сложной детали на примитивные составляющие). С другой стороны, именно возможность рекурсии делает реализацию дедуктивной СУБД очень сложной и во многих случаях неразрешимой эффективно проблемой.

Мы не будем здесь более подробно рассматривать конкретные проблемы, применяемые ограничения и используемые методы в дедуктивных системах. Отметим лишь, что обычно языки запросов и определения интенциональной части БД являются логическими (поэтому дедуктивные БД часто называют логическими). Имеется прямая связь дедуктивных БД с базами знаний (интенциональную часть БД можно рассматривать как БЗ). Более того, трудно провести грань между этими двумя сущностями; по крайней мере, общего мнения по этому поводу не существует.

Какова же связь дедуктивных БД с реляционными СУБД, кроме того, что реляционная БД является вырожденным частным случаем дедуктивной? Основным является то, что для реализации дедуктивной СУБД обычно применяется реляционная система. Такая система выступает в роли хранителя фактов и исполнителя запросов, поступающих с уровня дедуктивной СУБД. Между прочим, такое использование реляционных СУБД резко актуализирует задачу глобальной оптимизации запросов.

При обычном применении реляционной СУБД запросы обычно поступают на обработку по одному, поэтому нет повода для их глобальной (межзапросной) оптимизации. Дедуктивная же СУБД при выполнении одного запроса пользователя в общем случае генерирует пакет запросов к реляционной СУБД, которые могут оптимизироваться совместно.

Конечно, в случае, когда набор правил дедуктивной БД становится велик, и их невозможно разместить в оперативной памяти, возникает проблема управления их хранением и доступом к ним во внешней памяти. Здесь опять же может быть применена реляционная система, но уже не слишком эффективно. Требуются более сложные структуры данных и другие условия выборки. Известны частные попытки решить эту проблему, но общего решения пока нет.