Часть I. Распределенные системы управления базами данных

СУБД в архитектуре "клиент-сервер"

Лекция 1. Архитектура "клиент-сервер"

Открытые системы

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

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

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

Практической опорой системных и прикладных программных средств открытых систем является стандартизованная операционная система. В настоящее время такой системой является UNIX. Фирмам-поставщикам различных вариантов ОС UNIX в результате длительной работы удалось прийти к соглашению об основных стандартах этой операционной системы. Сейчас все распространенные версии UNIX в основном совместимы по части интерфейсов, предоставляемых прикладным, а в большинстве случаев и системным программистам. По мнению многих специалистов, несмотря на появление претендующей на стандарт системы Windows NT, возможно именно UNIX останется основой открытых систем в ближайшие годы.

Технологии и стандарты открытых систем обеспечивают реальную и проверенную практикой возможность производства системных и прикладных программных средств со свойствами мобильности (portability) и интероперабельности (interoperability). Свойство мобильности означает сравнительную простоту переноса программной системы в широком спектре аппаратно-программных средств, соответствующих стандартам. *Интероперабельность* означает упрощения комплексирования новых программных систем на основе использования готовых компонентов со стандартными интерфейсами.

Использование подхода открытых систем выгодно и производителям, и пользователям. Прежде всего, открытые системы обеспечивают естественное решение проблемы поколений аппаратных и программных средств. Производители таких средств не решают все проблемы заново. Они могут, по крайней мере, временно продолжать

комплексировать системы, используя существующие компоненты. Следует заметить, что при этом возникает новый уровень конкуренции. Все производители обязаны обеспечить некоторую стандартную среду, но вынуждены добиваться ее как можно лучшей реализации. Конечно, через какое-то время существующие стандарты начнут играть роль сдерживания прогресса, и тогда их придется пересматривать.

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

Клиенты и серверы сетей

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

Развитие этой идеи приводит к функциональному выделению компонентов сети: разумно иметь не только доступ к ресурсами удаленного компьютера, но также получать от этого компьютера комплекс сервисов, которые специфичны для ресурсов этого компьютера. При этом программные средства, поддерживающие эти сервисы нецелесообразно дублировать в нескольких узлах сети. Так мы приходим к различению рабочих станций и серверов сети.

Рабочая станция предназначена для непосредственной работы пользователя или категории пользователей и обладает ресурсами, соответствующими локальным потребностям данного пользователя. Специфическими особенностями рабочей станции могут быть:

- объем оперативной памяти (далеко не все категории пользователей нуждаются в наличии большой оперативной памяти);
- наличие и объем дисковой памяти (достаточно популярны бездисковые рабочие станции, использующие внешнюю память дискового сервера);
- характеристики процессора и монитора (некоторым пользователям нужен мощный процессор, других в большей степени интересует разрешающая способность монитора;
- для третьих обязательно требуются мощные средства для работы с графикой и т.д.).

При необходимости можно использовать ресурсы и/или услуги (сервисы), предоставляемые сервером.

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

Примерами серверов могут служить:

- сервер телекоммуникаций, обеспечивающий услуги по связи данной локальной сети с внешним миром;
- вычислительный или функциональный сервер, дающий возможность производить вычисления, которые невозможно выполнить на рабочих станциях;

- дисковый сервер, обладающий расширенными ресурсами внешней памяти и предоставляющий их в использование рабочим станциями и, возможно, другим серверам;
- файловый сервер, поддерживающий общее хранилище файлов для всех рабочих станций;
- сервер баз данных фактически обычная СУБД, принимающая запросы по локальной сети и возвращающая результаты;
- и другие.

Сервер компьютерной сети предоставляет ресурсы (услуги) рабочим станциям и/или другим серверам. Принято называть клиентом компьютерных сети, запрашивающий сервис у некоторого сервера и сервером - компонент компьютерной сети, оказывающий сервис некоторым клиентам.

Технология работы в архитектуре "клиент-сервер"

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

В этом разделе мы посмотрим на базу данных с несколько иной точки зрения. Общая цель систем баз данных - это, конечно, поддержка разработки и выполнения приложений баз данных. Поэтому на высоком уровне систему баз данных можно рассматривать как систему с очень простой структурой, состоящей из двух частей - *сервера* (машины базы данных, называемую сервером баз данных) и набора *клиентов* (или внешнего интерфейса).

- **Сервер** - это собственно СУБД. Он поддерживает все основные функции СУБД: определение данных, обработку данных, защиту и целостность данных и т.д. В частности, он предоставляет полную поддержку на внешнем, концептуальном и внутреннем уровнях. Поэтому "сервер" в этом контексте - это просто другое имя СУБД.
- **Клиент** - это различные приложения, которые выполняются "над" СУБД: приложения, написанные пользователями, и встроенные приложения, предоставляемые поставщиками СУБД или некоторыми сторонними поставщиками программного обеспечения. Конечно, с точки зрения пользователей, нет разницы между встроенными приложениями и приложениями, написанными пользователем, - все они используют один и тот же интерфейс сервера, а именно интерфейс внешнего уровня.

Исключениями являются специальные "служебные" приложения, которые называются утилитами. Такие приложения иногда могут работать только непосредственно на внутреннем уровне системы. Утилиты скорее относятся к непосредственным компонентам СУБД, чем к приложениям в обычном смысле. В нижеследующем разделе данной лекции утилиты обсуждаются более подробно. Приложения, в свою очередь, можно классифицировать на несколько четко определенных категорий.

1. *Приложения, написанные пользователями.* Это в основном профессиональные прикладные программы, написанные либо на общепринятом языке программирования, таком как С или PASCAL, либо на некоторых оригинальных языках, таких как FOCUS, хотя в обоих случаях эти языки должны как-то связываться с соответствующим подъязыком данных.

2. *Приложения, предоставляемые поставщиками* (часто называемые инструментальными средствами). В целом назначение таких средств - содействовать в процессе создания и выполнения других приложений, т.е. приложений, которые делаются специально для некоторой специфической задачи (хотя созданные приложения могут и не выглядеть как приложения в общепринятом смысле). Действительно, эта категория инструментальных средств позволяет пользователям, особенно конечным, создавать приложения без написания традиционных программ. Например, одно из предоставляемых поставщиками инструментальных средств может быть процессором языка запросов, с помощью которого конечный пользователь может выдавать незапланированные запросы к системе. Каждый такой запрос является, по существу, не чем иным, как специальным приложением (например, ISQL СУБД MS SQL Server), предназначенным для выполнения некоторых специфических функций.

Поставляемые инструментальные средства, в свою очередь, делятся на несколько самостоятельных классов:

- процессоры языков запросов;
- генераторы отчетов;
- графические бизнес-подсистемы;
- электронные таблицы:
- процессоры обычных языков;
- средства управления копированием;
- генераторы приложений;
- другие средства разработки приложений, включая CASE-продукты (CASE или Computer-Aided Software Engineering - автоматизация разработки программного обеспечения), и т.д.

Подробности об этих приложениях выходят за рамки данного курса, однако следует отметить, что главная задача системы баз данных - это поддержка создания и выполнения приложений, поэтому качество имеющихся клиентских инструментальных средств должно быть главным образом при выборе базы данных (т.е. процессе выбора подходящей системы для данного заказчика). Другими - словами, СУБД сама но себе не единственный и - не обязательно важнейший фактор, который нужно учитывать.

Необходимо отметить, что так как система в целом может быть четко разделена на две части (сервер и клиенты), появляется возможность работы этих двух частей на разных машинах. Иначе говоря, существует возможность распределенной обработки.

Распределенная обработка предполагает, что отдельные машины можно соединить какой-нибудь коммуникационной сетью таким способом, что определенная задача, обрабатывающая данные, может быть распределена на нескольких машинах в сети. На самом деле, эта возможность настолько заманчива по различным соображениям, главным образом практическим, что термин "клиент/сервер" стал применяться исключительно в случае, если сервер и клиенты действительно находятся на разных машинах. Такое применение термина - небрежное, но очень распространенное. Технология, поддерживающая распределённую обработку данных должна обеспечивать клиенту доступ к распределённой БД точно так же, как доступ к централизованной БД. При этом данные могут храниться на локальном узле, на удалённом узле или обоих узлах - их расположение должно оставаться *прозрачным* как для конечного пользователя, так и для программы.

Прежде чем рассмотреть упрощенную технологию работы в клиент-серверной архитектуре, приведем историю развития архитектуры обработки данных с использованием баз данных. К настоящему времени их три:

А. База данных на мэйнфрейме.

B. Архитектура файл-сервер.
C. Архитектура клиент/сервер.

А. Исторически появилась первая. Извлечение данных и обработка происходит на одной машине. Однопользовательская система, ее пример приведен на рис. 1.1.

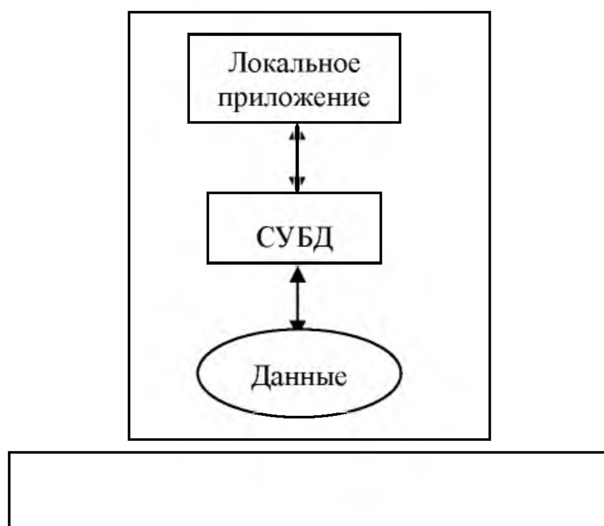


Рис. 1.1. Схема однопользовательской системы

В. С появлением сетей данные стали хранить на файл-сервере. Это первый вид многопользовательской системы. В этом случае их поиск и обработка происходит на рабочих станциях (рис. 1.2). При таком подходе на рабочую станцию посылаются не только данные, необходимые конечному пользователю, но и данные, которые используются только для выполнения запроса (например, фрагменты индексных файлов или данные, которые будут отброшены при выполнении запроса). Таким образом, объём "лишней" информации зачастую превышает объём "нужной".

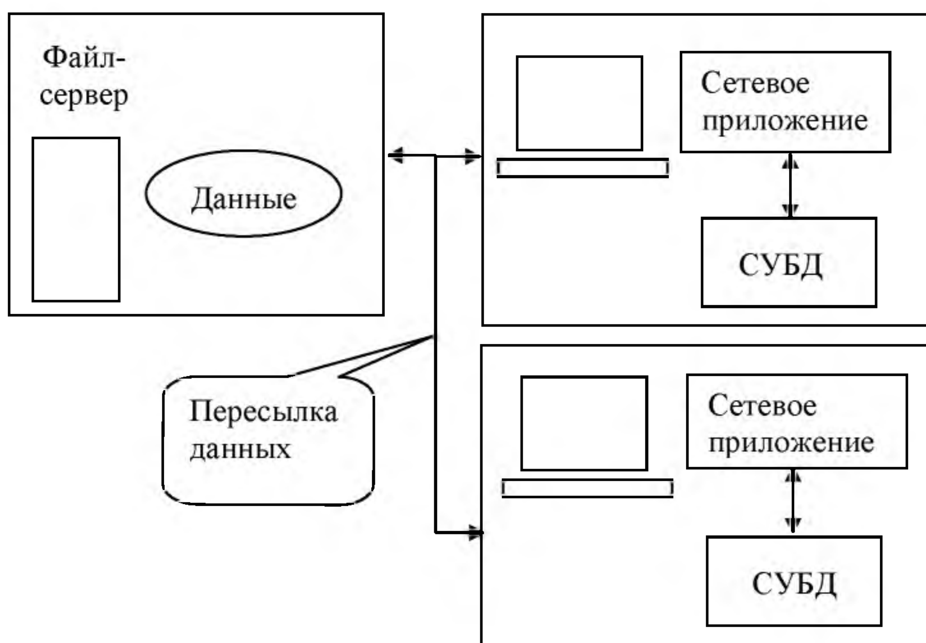


Рис. 1.2. Схема файл-серверной архитектуры

Время реакции на запрос пользователя складывается из времени передачи данных с файл-сервера на рабочую станцию и времени выполнения запроса на рабочей станции. Чтобы время реакции такой системы было приемлемым, надо ускорить обмен данными с диском и нарастить объём оперативной памяти для кэширования данных с диска. Также

желательно в качестве рабочей станции использовать мощный компьютер. Узким местом может оказаться сетевая среда, поэтому пропускная способность сетевой шины - тоже немаловажный показатель. Если растёт число одновременно работающих пользователей и объём хранимой информации, размер пересылаемой информации растёт, т.е. увеличивается сетевой трафик. И как результат, время реакции системы значительно падает. Такая технология подразумевает, что на каждой рабочей станции находится свой экземпляр СУБД, работающий с одними и теми же данными. Взаимодействие этих СУБД для синхронизации работы через промежуточное звено в виде файл-сервера приводит к дополнительным потерям.

С. И последний вид архитектуры, архитектура клиент/сервер. Характеризуется наличием одной СУБД для всех пользователей, которая расположена на сервере. При такой технологии программа пользователя (клиента) формирует запрос на отбор данных и отсылает запрос к серверу. Сервер отбирает данные, соответствующие выполняемому запросу, и отсылает их программе-клиенту (приложению). Программа-клиент обрабатывает полученные данные и предоставляет их пользователю. В этом случае объём передаваемой информации, а значит и сетевой трафик значительно ниже, чем при использовании файл-сервера. Логично было бы ожидать, что общее время отклика должно сократиться.

Однако время реакции в такой системе складывается из времени передачи запроса, времени ожидания ресурсов на сервере (например, процессора или дисковой операции), времени выполнения запроса и времени передачи результатов на рабочую станцию - программе-клиенту. Причём время ожидания на сервере может съедать львиную долю общего времени выполнения запроса. При разработке программ, работающих по технологии клиент-сервер, необходимо учитывать это и не обращаться к серверу за одной записью, а читать данные “пачками”.

Если сетевой трафик уменьшается, то узким местом становится компьютер, выполняющий роль сервера. Требования к нему очень высоки. В качестве сервера необходимо выбирать мощный компьютер, но мощность рабочих станций при этом увеличивать не обязательно.

В настоящее время различают две модели архитектуры клиент/сервер – двухуровневую и трехуровневую. Для двухуровневой модели характерна ситуация, когда БД состоит из таблиц локальных БД, которые находятся на одном узле, и там же функционируют сервер БД, прикладные программы выполняются на клиентских узлах (рис. 1.3).

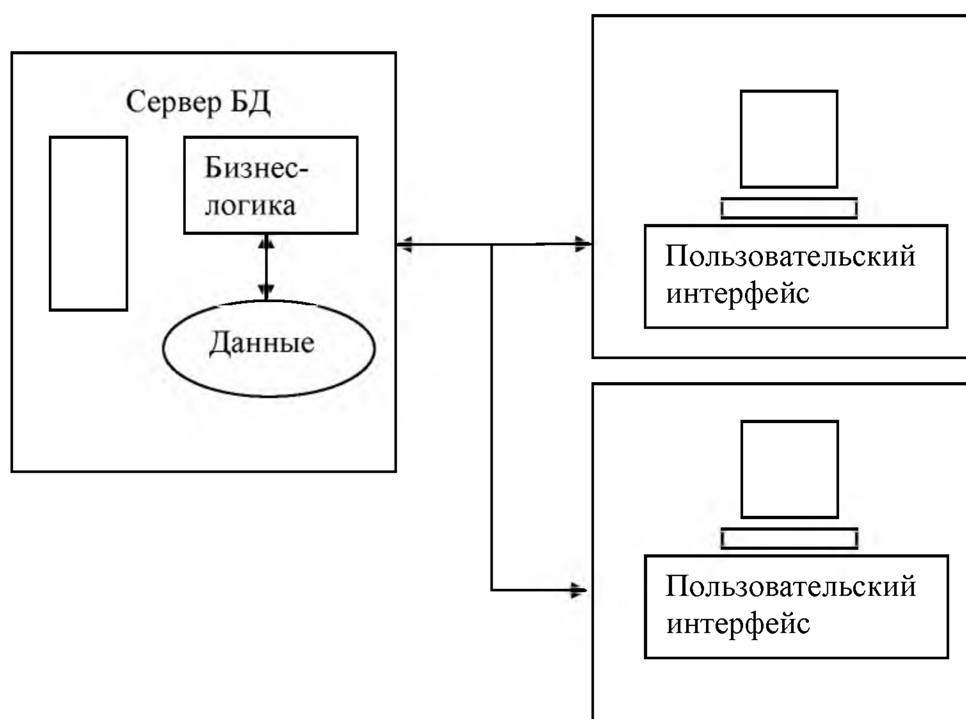


Рис. 1.3. Двухуровневая модель архитектуры клиент/сервер

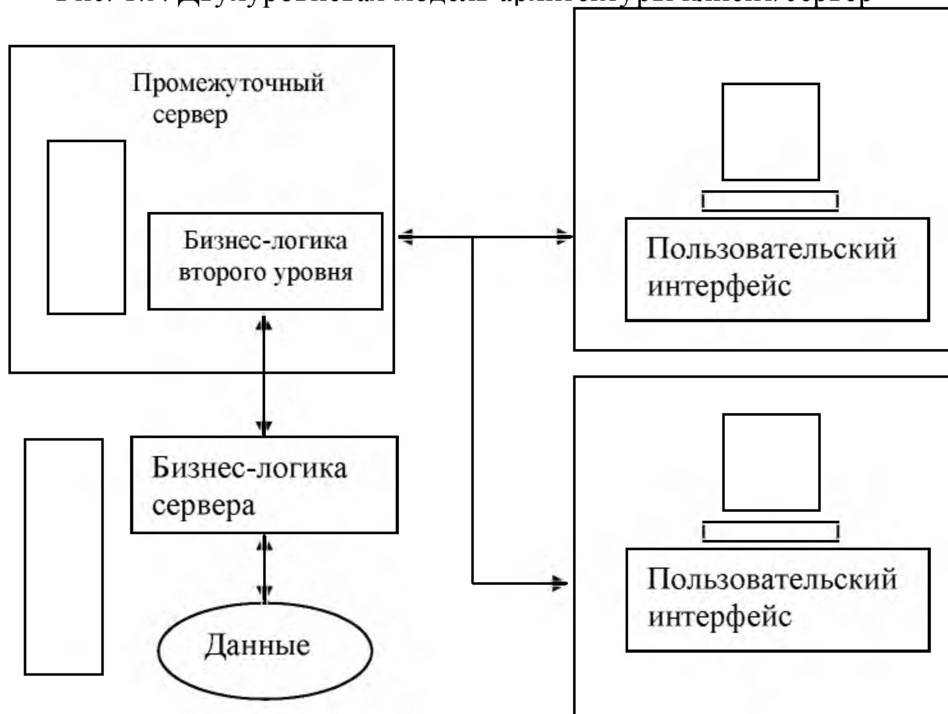


Рис. 1.4. Трехуровневая модель архитектуры клиент/сервер

Для трёхуровневой модели (рис. 1.4) характерна ситуация, когда распределенная БД состоит из таблиц локальных БД, которые находятся на одном узле, программы доступа к данным и часть прикладных программ находятся на другом узле (возможно на сервере приложений), а клиентские приложения на клиентских узлах (возможно только внешний интерфейс).

Для обеих моделей возможна ситуация, когда БД состоит из локальных БД, которые находятся на удаленных узлах (возможно географически удаленных), тогда такая база данных называется распределенной и обязателен сервер БД.

Сервер распределенной БД (Distributed DataBase Server) - это небольшая операционная система, которая должна решать как минимум следующие задачи:

- управление именами в распределённой среде (глобальный словарь данных);
- оптимизация распределённых запросов;
- управление распределёнными транзакциями.

Принципы взаимодействия между клиентскими и серверными частями

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

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

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

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

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

Преимущества протоколов удаленного вызова процедур

Протоколы удаленного вызова процедур особенно важны в системах управления базами данных, основанных на архитектуре "клиент-сервер".

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

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

Типичное разделение функций между клиентами и серверами

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

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

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

Требования к аппаратным возможностям и базовому программному обеспечению клиентов и серверов

Из предыдущих рассуждений видно, что требования к аппаратуре и программному обеспечению клиентских и серверных компьютеров различаются в зависимости от вида использования системы.

Если разделение между клиентом и сервером достаточно жесткое (как в большинстве современных СУБД), то пользователям, работающим на рабочих станциях или персональных компьютерах, абсолютно все равно, какая аппаратура и операционная система работают на сервере, лишь бы он справлялся с возникающим потоком запросов. Но если могут возникнуть потребности перераспределения функций между клиентом и сервером, то уже совсем не все равно, какие операционные системы используются.

Утилиты

Утилиты - это программы, разработанные для администратора баз данных и используемые при выполнении различных административных задач. Как уже упоминалось выше, некоторые утилиты выполняются на внешнем уровне системы и потому представляют собой не что иное, как приложения специального назначения. Некоторые из них могут предоставляться даже не поставщиками СУБД, а скорее некоторыми сторонними поставщиками программного обеспечения. Однако другие утилиты выполняются непосредственно на внутреннем уровне (т.е. действительно являются частью сервера) и поэтому должны предоставляться поставщиками СУБД. Ниже приводится несколько типичных примеров типов утилит, которые часто применяются на практике.

- *Процедуры загрузки*, применяемые для создания первоначальной версии базы данных из одного или более файлов, которые не принадлежат базе данных.
- *Процедуры выгрузки-перезагрузки*, применяемые для выгрузки базы данных или порции из нее, дублирования памяти с целью восстановления и перезагрузки данных из таких дублированных копий (конечно, "утилита перезагрузки", по существу, идентична рассмотренной утилите загрузки).
- *Процедуры реорганизации,* применяемые для перераспределения данных в базе данных по различным соображениям (обычно выполняется для повышения производительности), например, для группировки некоторых данных каким-то определенным способом на диске или освобождения пространства, занятого данными, которые больше не используются.
- *Статистические процедуры*, применяемые для вычисления различных статистических показателей производительности, таких как распределение размеров файлов или значений данных и счетчиков ввода-вывода и т.п.
- *Процедуры анализа,* применяемые для анализа только что упомянутой статистики.

Этот список охватывает лишь небольшую часть функций, предоставляемых утилитами. Кроме перечисленных видов функций, существует множество других.

Распределенная обработка

Как мы определили выше, термин “распределенная обработка” означает, что разные машины можно соединить в коммуникационную сеть так, что одна задача обработки данных распределяется между несколькими машинами в сети. Термин "параллельная обработка" используется практически с тем же значением, за исключением того, что разные машины с физической точки зрения расположены близко друг к другу в "параллельных" системах. Это вовсе не обязательно в "распределенной" системе, например, они могут быть удалены географически. Связь между различными машинами осуществляется с помощью специального программного обеспечения для управления сетью. Распределенная обработка может быть самой разнообразной и осуществляться на разных уровнях. Как отмечалось выше, в одном из простых случаев запускается сервер СУБД на одной машине и клиентское приложение на другой. Термин "клиент/сервер" фактически стал синонимом структуры, изображенной на рис.1.5, в соответствии с которой клиент и сервер запускаются на разных машинах. В действительности существует множество аргументов в пользу такой схемы.

- Первый аргумент связан с параллельной обработкой, а именно: в этом случае для всей задачи применяется несколько процессоров и обработка сервера (базы данных) и клиента (приложения) осуществляется параллельно. Поэтому время ответа и производительное время должны уменьшиться.

- Машина сервера может быть изготовлена по специальному заказу, приспособлена для работы с СУБД ("машина базы данных") и может обеспечить лучшую производительность СУБД.
- Машина клиента может быть персональной станцией, приспособленной к потребностям конечного пользователя, и поэтому обеспечивать лучший интерфейс, полное соответствие требованиям, быструю реакцию и в целом дополнительные удобства при использовании.
- Несколько разных машин клиентов могут иметь доступ к одной и той же машине сервера. Поэтому одна база данных может совместно использоваться несколькими отдельными клиентскими системами (рис. 1.5).

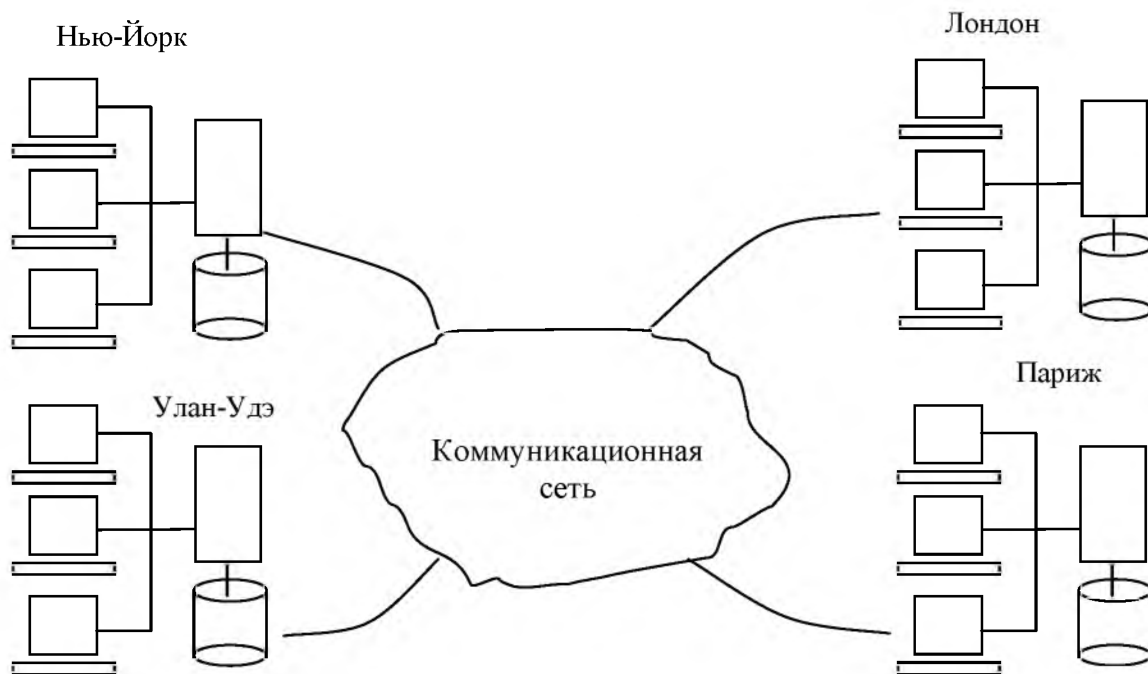


Рис. 1.5. Типичная распределенная система баз данных

К сказанному выше можно добавить еще одно преимущество выполнения сервера и клиента на отдельных машинах - соответствие практической работе многих предприятий. Это довольно распространенный способ для отдельного предприятия. Например, банк работает со многими компьютерами, сохраняющими данные для одной части предприятия на одном компьютере, а данные для другой части - на другом. Это также очень распространено среди пользователей, которым необходим, по крайней мере иногда, доступ с одного компьютера к данным, хранимым на другом компьютере. Следуя примеру банка, можно сказать, что весьма вероятно, пользователям одного отделения банка будет иногда необходим доступ к данным, сохраняемым в другом отделении. Следовательно, машины клиентов могут иметь свои собственные сохраняемые данные, а машина сервера может иметь свои собственные приложения. Поэтому, вообще говоря, каждая машина будет выступать в роли сервера для одних пользователей и в роли клиента для других; иными словами, каждая машина будет поддерживать полную систему баз данных.

Отметим последнее преимущество, которое состоит в том, что отдельная машина клиента может иметь доступ к нескольким разным машинам серверов. Это полезная возможность, поскольку, как уже упоминалось, предприятие обычно выполняет обработку данных таким образом, что полный набор всех данных не сохраняется на одной машине, а распределяется на отдельных машинах, а для приложений иногда необходим доступ к данным нескольких машин. Такой доступ в основном предоставляется двумя

способами.

- Клиент может получать доступ к любому количеству серверов, но лишь к одному в одно и то же время (т.е. каждый запрос к базе данных должен быть направлен только к одному серверу). В такой системе невозможно за один запрос получить комбинированные данные двух или более серверов. Кроме того, пользователь в такой системе должен знать, на какой именно машине какая часть данных содержится.
- Клиент может получать доступ к любому количеству серверов одновременно (т.е. за один запрос можно получить комбинированные данные двух или более серверов). В этом случае серверы рассматриваются клиентом как один (с логической точки зрения), и пользователь не обязан знать, на какой именно машине какая часть данных содержится.

Второй случай - это пример системы, которую обычно называют ***распределенной системой баз данных***. Тема распределенных баз данных сама по себе весьма обширна. Подводя ее определение к логическому завершению, отметим следующее: полная поддержка для распределенных баз данных означает, что отдельное приложение может "прозрачно" обрабатывать данные, распределенные на множестве различных баз данных, управление которыми осуществляют разные СУБД, работающие на многочисленных машинах с различными операционными системами, соединенных вместе коммуникационными сетями. Здесь понятие "прозрачно" означает, что приложение выполняет обработку данных с логической точки зрения, как будто управление данными полностью осуществляется одной СУБД, работающей на отдельной машине. Такая возможность может показаться невероятно трудной задачей, но весьма желанной с практической точки зрения. Подробнее распределенные базы данных рассматривается далее.

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

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

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

Основной проблемой систем, основанных на архитектуре "клиент-сервер", является то, что в соответствии с концепцией открытых систем от них требуется мобильность в как можно более широком классе аппаратно-программных решений открытых систем. Даже если ограничиться UNIX-ориентированными локальными сетями, в разных сетях применяется разная аппаратура и протоколы связи. Попытки создания систем, поддерживающих все возможные протоколы, приводит к их перегрузке сетевыми деталями в ущерб функциональности.

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

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

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

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

Распределенные базы данных

Лекция 2. Распределенные БД

Основная задача систем управления распределенными базами данных состоит в обеспечении средства интеграции локальных баз данных, располагающихся в некоторых узлах вычислительной сети, с тем, чтобы пользователь, работающий в любом узле сети, имел доступ ко всем этим базам данных как к единой базе данных.

При этом должны обеспечиваться:

- простота использования системы;
- возможности автономного функционирования при нарушениях связности сети или при административных потребностях;
- высокая степень эффективности.

Фундаментальный принцип. *Для пользователя распределенная система должна выглядеть точно так же, как НЕраспределенная система.*

Иначе говоря, работу пользователей в распределенной системе следует организовать таким же образом, как если бы она не была распределенной. Все связанные с распределенными системами проблемы являются (или должны быть) внутренними и должны возникать только на внутреннем уровне или уровне разработки, а не на внешнем уровне или на уровне пользователей.

В данном случае термин "пользователи" относится к пользователям (потребителям или разработчикам приложений), которые выполняют операции *управления данными*. Все эти операции должны оставаться логически неизменными, в отличие от операций *определения данных*, которые, наоборот, могут быть расширены в распределенной системе. Например, пользователь на узле *X* может указать, что хранимое отношение можно разделить на "фрагменты" для физического хранения на узлах *Y* и *Z*.

Изложенный фундаментальный принцип приводит к набору вспомогательных правил и целей.

Цели или правила распределенных систем

Термин "правило" вместе с разъяснением смысла этого понятия было впервые предложен К. Дж. Дейтом, а понятие "фундаментальный принцип" был им назван "правилом нуль" (Rule Zero). Однако далее вместо термина "правила" будет использоваться более подходящий и менее догматичный термин "цель". Их всего двенадцать.

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.

Эти двенадцать целей не являются независимыми одна от другой, к тому же не все они равнозначны. Различные пользователи могут придавать разное значение разным цепям в разном окружении. Ими также не исчерпывается список всех возможных целей.

Однако они весьма полезны для понимания основ распределенной технологии и для общей характеристики функциональности некоторой распределенной системы. Рассмотрим краткий обзор каждой из этих целей.

Следует отметить, что, необходимо различать распределенные системы и системы, в которых поддерживаются некоторые способы удаленного доступа к данным (безусловно, поддерживаемого в системах клиент/сервер). В системах с "*удаленным доступом к данным*" пользователь может одновременно работать с данными, расположенными на нескольких удаленных узлах, однако в таком случае будут видны соединения между ними. Следовательно, пользователь в большей или меньшей мере будет знать о том, что работа ведется с удаленными данными, и оперировать с ними соответствующим образом. В истинной распределенной системе, наоборот, все соединения от пользователя скрыты.

1. Локальная автономия

В распределенной системе узлы следует делать автономными. Локальная автономия означает, что операции на данном узле управляются этим узлом, т.е. функционирование любого узла X не зависит от успешного выполнения некоторых операций на каком-то другом узле. В противном случае может возникнуть крайне нежелательная ситуация, а именно: выход из строя узла Y может привести к невозможности исполнения операций на узле X, даже если с узлом X ничего не случилось. Из принципа локальной автономии также следует, что владение и управление данными осуществляется локально вместе с локальным ведением учета. Действительно, даже если доступ к данным осуществляется с других удаленных узлов, все они относятся к некоторой локальной базе данных. Такие вопросы, как безопасность, целостность и структура хранения локальных данных, остаются под контролем и юрисдикцией этого локального узла.

В действительности цель локальной автономии достигается не полностью, поскольку есть множество ситуаций, в которых узел X должен предоставить некоторую часть управления другому узлу Y. Поэтому цель достижения локальной автономии требует более точной формулировки, а именно: *узлы следует делать автономными в максимально возможной степени.*

2. Независимость от центрального узла

Под локальной автономией подразумевается, что *все узлы должны рассматриваться как равные*. Следовательно, не должно существовать никакой зависимости и от центрального "основного" узла с некоторым централизованным обслуживанием, например централизованной обработкой запросов, централизованным управлением транзакциями или централизованным присвоением имен. Таким образом, вторая цель является логическим следствием первой (если достигнута первая цель, то вторая - тем более). Однако, если локальная автономия не достигается в полной мере, даже достижение независимости от центрального узла само по себе очень важно и может рассматриваться как отдельная цель.

Зависимость от центрального узла нежелательна по крайней мере по двум причинам. Во-первых, центральный узел может быть "*узким*" местом всей системы, а во-вторых, более важно то, что система в таком случае становится *уязвимой*, т.е. при повреждении центрального узла может выйти из строя вся система.

3. Непрерывное функционирование

Одним из основных преимуществ распределенных систем является то, что они обеспечивают более высокую *надежность* и *доступность*.

- **Надежность** - вероятность того, что система исправна и работает в любой заданный момент. Надежность повышается благодаря работе распределенных систем не по принципу "все или ничего", а в постоянном режиме. Это означает, что работа системы продолжается, хотя и на более низком уровне, даже в случае

неисправности некоторого отдельного компонента, например отдельного узла.

- **Доступность** - вероятность того, что система исправна и работает в течение некоторого промежутка времени. Доступность повышается частично по той же причине, а частично благодаря возможности репликации данных (подробнее это описывается ниже).

Приведенные выше рассуждения применимы также для случая незапланированного выключения некоторого компонента внутри системы, например вследствие какой-либо неисправности. Незапланированные выключения крайне нежелательны, но их возникновения очень трудно избежать. В идеальном случае следовало бы вовсе исключить запланированные выключения, т.е. не выключать систему при выполнении любых операций, например при добавлении нового узла или обновлении версии СУБД на некотором узле.

4. Независимость от расположения

Основная идея независимости от расположения (которая также называется ***прозрачностью расположения***) достаточно проста: *пользователям не следует знать, в каком физическом месте хранятся данные, наоборот, с логической точки зрения пользователям следовало бы обеспечить такой режим, при котором создается впечатление, что все данные хранятся на их собственном локальном узле*. Обеспечить независимость от расположения весьма желательно, поскольку при этом существенно упрощаются пользовательские программы и терминальная деятельность. В частности, это позволяет осуществлять миграцию данных от узла к узлу без объявления недействительными любых пользовательских программ и видов терминальной деятельности. Процесс миграции весьма полезен, поскольку позволяет перемешаться данным по всей сети в ответ на изменение требований к производительности.

Нетрудно заметить, что принцип независимости от расположения - это не что иное, как расширение обычной концепции (физической) независимости данных, применяемой для распределенной обработки данных. Следует сказать, что каждая цель из приведенного выше списка, в название которой входит слово "независимость", может рассматриваться как некоторое расширение понятия независимости данных.

5. Независимость от фрагментации

В системе поддерживается фрагментация данных, если некое хранимое отношение в целях физического хранения можно разделить на части, или "фрагменты". Фрагментация желательна для повышения производительности системы, поскольку данные лучше хранить в том месте, где они наиболее часто используются. При такой организации многие операции будут чисто локальными, а объем пересылаемых вести данных снизится. Например, рассмотрим отношение сотрудников EMP, показанное на рис.2.1. В системе, в которой поддерживается фрагментация, определены два фрагмента, показанных в нижней части рис.2.1.

Предполагается, что кортежи сотрудников некоторым заданным образом непосредственно отображаются на структуру физического хранения, D1 и D3 являются отделами в Нью-Йорке (New York), а D2 - в Лондоне (London). Иначе говоря, в Нью-Йорке будут храниться кортежи сотрудников из Нью-Йорка, а в Лондоне - кортежи сотрудников из Лондона. Обратите внимание на внутренние системные имена фрагментов: N_EMP и L_EMP.

Существует два основных типа фрагментации - *горизонтальная и вертикальная*, которые связаны с реляционными операциями выборки и проекции соответственно. На рис.2.1 показана горизонтальная фрагментация. Иначе говоря, фрагментом может быть *любое произвольное подчиненное отношение*, которое можно вывести из исходного отношения с помощью операций выборки и проекции. При этом следует учесть приведенные ниже допущения.

- Предполагается без утраты общности, что все фрагменты данного отношения *независимы*, т.е. ни один из фрагментов не может быть выведен из других фрагментов либо иметь выборку или проекцию, которая может быть выведена из других фрагментов. Если есть необходимость сохранить одну и ту же информацию в нескольких разных местах, для этого следует использовать механизм репликации системы.
- Проекции не должны допускать потерю информации.

Восприятие пользователя

EMP#	DEPT#	SALARY
E1	D1	40K
E2	D2	42K
E3	D3	30K
E4	D4	35K
E5	D5	48K

Нью-Йорк

N_EMP

EMP#	DEPT#	SALARY
E1	D1	40K
E2	D1	42K
E5	D3	48K

Лондон

N_EMP

EMP#	DEPT#	SALARY
E3	D2	30K
E4	D2	35K

Рис. 2.1. Пример фрагментации

Реконструкцию исходного отношения на основе его фрагментов можно осуществить с помощью операций соединения (для вертикальных фрагментов) и объединения (для горизонтальных фрагментов). Обратите внимание, что в случае объединения не потребуется выполнять операцию исключения дубликатов (*самостоятельно*).

В отношении вертикальной фрагментации необходимо привести дополнительные разъяснения. Как уже упоминалось выше, при такой фрагментации не должна допускаться потеря информации, потому что фрагментация отношения EMP на проекции (EMP#, DEPT#) и (SALARY) не будет достоверной. Однако в некоторых системах хранимые отношения имеют скрытый атрибут TID ("tuple ID" - идентификатор кортежа), т.е. физический или логический адрес этого кортежа. Атрибут TID является потенциальным ключом для данного кортежа, поэтому если бы отношение EMP содержало такой атрибут, то оно могло бы быть корректно фрагментировано на проекции (TID, EMP#, DEPT#) и (TID, SALARY), поскольку такая фрагментация происходит без потери информации. Обратите внимание, что именно легкость выполнения фрагментации и реконструкции - одна из многих причин, по которым для распределенных систем используется реляционная модель. Дело в том, что в ней предусмотрены именно те операции, которые необходимы для выполнения таких задач.

Наконец, переходя к главному, нужно сказать, что в системе, поддерживающей фрагментацию данных, следует также предусмотреть поддержку независимости от фрагментации (или прозрачность фрагментации). Иными словами, с логической точки зрения пользователям следует обеспечить такой режим работы, при котором данные кажутся нефрагментированными. Независимость от фрагментации (как и независимость от расположения) весьма желательна, поскольку упрощает пользовательские программы и терминальную деятельность. В частности, она позволяет в любой момент осуществить дефрагментацию данных (и фрагменты могут быть перераспределены в любой момент времени) в ответ на изменение требований к производительности, причем без необходимости отключения любых пользовательских программ и терминальной

деятельности. Независимость от фрагментации предполагает, что данные будут представлены для пользователей в виде логически комбинированных фрагментов на основе соответствующих объединений и соединений. При этом системный оптимизатор отвечает за определение фрагментов, к которым необходимо обеспечить физический доступ для выполнения любого заданного пользователем запроса. В качестве примера допустим, что для фрагментации, показанной на рис. 2.1, пользователь задает следующий запрос:

EMP WHERE SALARY > 40К AND DEPT# = 'DI'

В таком случае оптимизатор системы будет знать из определений фрагментов (которые хранятся в системном каталоге), что так как все результаты могут быть получены на узле в Нью-Йорке, то нет никакой необходимости осуществлять доступ к узлу в Лондоне. Рассмотрим этот же пример более тщательно. Отношение EMP воспринимается пользователем как представление базовых фрагментов N_EMP и L_EMP

EMP = N_EMP UNION L_EMP

Таким образом, оптимизатор преобразует исходный запрос пользователя в следующий:

(N_EMP UNION L_EMP) WHERE SALARY > 40K AND DEPT# = 'D1'

А это выражение, в свою очередь, можно преобразовать к виду

(N_EMP WHERE SALARY > 40K AND DEPT# = 'D1'
UNION
(N_EMP WHERE SALARY > 40K AND DEPT# = 'D1')

Из хранящегося в каталоге определения фрагмента L_EMP оптимизатору известно о том, что второе из этих двух выражений даст в результате пустое отношение (условие целостности DEPT# = 'D1' AND DEPT# = 'D2' не может быть оценено как истинное). Таким образом, все выражение в целом можно свести к следующему упрощенному виду:

N_EMP WHERE SALARY > 40K AND DEPT# = 'DI '

Теперь оптимизатору известно, что доступ требуется осуществить только к узлу в Нью-Йорке. *Самостоятельно рассмотрите действия, выполняемые оптимизатором при работе с приведенным ниже запросом*.

EMP WHERE SALARY > 40K

Как следует из сказанного выше, проблема поддержки операций для фрагментированных отношений имеет некоторое сходство с проблемой поддержки операций для объединенных и соединенных представлений. Фактически, это разные проявления одной и той же проблемы при рассмотрении общей архитектуры системы с разных точек зрения. В частности, проблема обновления фрагментированных отношений- это проблема обновления в объединениях и соединениях представлений. Отсюда также следует, что при обновлении заданный кортеж может мигрировать из одного фрагмента в другой при условии, что он больше не удовлетворяет предикату отношения для фрагмента, к которому относился до обновления.

6. *Независимость от репликации*

В системе поддерживается независимость от репликации, если заданное хранимое отношение или заданный фрагмент могут быть представлены несколькими различными копиями, или репликами, хранимыми на нескольких различных узлах. Ниже приведен пример репликации, схематически показанный на рис. 2.2.

```
REPLICATE N_EMP
          LN_EMP AT SITE 'London' ;
REPLICATE L_EMP
```

NL_EMP AT SITE 'New York'

Обратите внимание на внутренние системные имена реплик NL_EMP и LN_EMP.

Нью-Йорк					*Лондон*
N_EMP			**L_EMP**		
EMP#	DEPT#	SALARY	EMP#	DEPT#	SALARY
E1	D1	40K	E3	D2	30K
E2	D1	42K	E4	D2	35K
E5	D3	48K			
NL_EMP			**LN_EMP**		
EMP#	DEPT#	SALARY	EMP#	DEPT#	SALARY
E3	D2	30K	E1	D1	40K
E4	D2	35K	E2	D1	42K
(NL_EMP – реплика L_EMP)			E5	D3	48K
			(LN_EMP – реплика N_EMP)		

Рис. 2.2. Пример репликации

Репликация полезна по крайней мере по двум причинам. Во-первых, благодаря ей достигается большая производительность (приложения могут работать с локальными копиями, не обмениваясь данными с удаленными узлами). Во-вторых, репликация также позволяет обеспечить большую доступность. Реплицированный объект остается доступным для обработки до тех пор, пока остается хотя бы одна его реплика. Главный недостаток репликации заключается в том, что при обновлении заданного реплицируемого объекта, все копии этого объекта также должны обновляться. Этот недостаток называется проблемой распространения обновления, которая будет более подробно описана ниже в этой главе.

Следует отметить, что репликация в распределенной системе представляет собой особый вид воплощения идеи управляемой избыточности.

В идеальном случае репликация, как и фрагментация, должна быть "прозрачной для пользователя". Иначе говоря, в системе, в которой поддерживается репликация данных, должна также поддерживаться независимость от репликации (или прозрачность репликации), т.е. пользователи, по крайней мере с логической точки зрения, должны работать в таком режиме, как будто данные не реплицированы вовсе. Независимость от репликации (так же как независимость от расположения и независимость от фрагментации) весьма желательна, поскольку позволяет существенно упростить пользовательские программы и терминальную деятельность. В частности, при этом разрешается в любой момент создавать и удалять реплики в ответ на изменения требований без отключения этих пользовательских программ и терминальной деятельности.

Независимость от репликации подразумевает, что системный оптимизатор отвечает за определение физического доступа именно к тем репликам, которые необходимы для удовлетворения заданного пользовательского запроса.

В заключение следует отметить, что многие коммерческие программные продукты в настоящее время предлагают такую форму репликации, которая не обеспечивает полной независимости от репликации (т.е. она не полностью "прозрачна для пользователя"). К этому вопросу мы еще вернемся при обсуждении проблемы распространения обновления ниже в этой главе.

7. Обработка распределенных запросов

Здесь рассматриваются два важных и обширных вопроса.

1. Рассмотрим запрос "Получить сведения о находящихся в Лондоне поставщиках

красных деталей", причем допустим, что пользователь находится в Нью-Йорке, а данные хранятся на узле в Лондоне. Предположим также, что такому запросу удовлетворяют *n* поставщиков. Если система является реляционной, то в запросе будут содержаться два сообщения - одно с запросом из Нью-Йорка для Лондона, а другое с возвращаемым результатом, т.е. набором из и кортежей, которые пересылаются из Лондона в Нью-Йорк. С другой стороны, если система не является реляционной, а работает в режиме последовательной обработки данных, то в запрос будет включено 2*n* сообщений - *n* из Нью-Йорка в Лондон с запросом сведений о "следующем" поставщике, а также *n* из Лондона в Нью-Йорк для возвращения сведений о "следующем" поставщике. Этот пример иллюстрирует, что реляционная система по производительности может на несколько порядков превосходить нереляционную (по крайней мере на уровне множеств).

2. Вопрос оптимизации более важен для распределенной, нежели для централизованной системы. Основная причина заключается в том, что для выполнения охватывающего несколько узлов запроса существует довольно много способов перемещения данных по сети. В таком случае чрезвычайно важно найти наиболее эффективную стратегию. Например, запрос на объединение отношения R_x, хранимого на узле *X*, и отношения R_y, хранимого на узле *Y*, мог бы быть выполнен с помощью перемещения отношения R_x на узел *Y*, перемещения отношения R_y на узел *X* или перемещения этих двух отношений на третий узел *Z* (и т.д.). Эта ситуация подробно рассматривается ниже, а здесь следует отметить, что на основе некоторых правдоподобных предположений можно составить шесть различных стратегий обработки такого запроса. При этом время отклика может варьироваться от минимального значения, равного одной десятой секунды, до максимального, равного приблизительно шести часам! Таким образом, оптимизация может оказаться весьма существенной, и этот факт, в свою очередь, может рассматриваться как еще одна причина, по которой распределенные системы всегда являются реляционными. Особенность заключается в том, что запросы могут быть оптимизированы на уровне множеств, но не на уровне записей.

8. Управление распределенными транзакциями

Существует два основных аспекта управления обработкой транзакций, а именно: управление восстановлением и управление параллелизмом, каждому из которых в распределенных системах должно уделяться повышенное внимание. Однако, прежде чем перейти к подробному описанию, следует ввести новое понятие "***агент***". В распределенной системе выполнение одной транзакции может быть связано с исполнением кода на нескольких узлах, в частности она может выполнять операции обновления на нескольких узлах. В таком случае говорят, что каждая транзакция состоит из *нескольких агентов, т.е. процессов, выполняемых на данном узле по указанию заданной транзакции.* Системе необходимо указать, что некоторые два агента представляют собой части одной и той же транзакции, например для того, чтобы избежать возникновения тупиковой ситуации для двух агентов, которые содержит одна и та же транзакция.

Несколько слов об управлении восстановлением. Для подтверждения того, что в распределенном окружении данная транзакция является атомарной (типа «все-или-ничего»), система должна убедиться, что агенты заданного набора должны быть все вместе либо завершены, либо отменены. Эта цель может быть достигнута только за счет реализации протокола двухфазной фиксации. Несколько более подробных замечаний на эту тему представлено ниже.

Что касается управления параллелизмом, то в распределенных системах, так же как и в нераспределенных, это управление обычно основано на блокировке. (В нескольких более современных системах уже используются средства многовариантного управления, однако в большинстве систем для этого все же продолжает использоваться метод блокировки). Более подробно эта тема рассматривается ниже.

9. Независимость от аппаратного обеспечения

Здесь трудно добавить что-либо существенное, поскольку основная идея фактически полностью изложена в названии этого раздела. Используемые в настоящее время компьютеры характеризуются весьма большим разнообразием, среди них можно встретить компьютеры фирм IBM, DEC, HP, а также персональные компьютеры и рабочие станции других типов. В связи с этим существует реальная необходимость интеграции данных на всех этих системах и создания для пользователя "представления единой системы". Таким образом, весьма важной является возможность запуска копий одной и той же СУБД на разном аппаратном обеспечении с тем, чтобы разные компьютеры могли работать в распределенной системе как равные партнеры.

10. Независимость от операционной системы

Эта цель является следствием предыдущей, и ее описание будет также немногословным. Очевидно, важно запустить одну и ту же СУБД не только на разном аппаратном обеспечении, но и на разных операционных системах, причем даже в тех случаях, когда разные операционные системы используются на однотипном аппаратном обеспечении. Например, для того чтобы версии СУБД для операционной системы MVS, а также для систем UNIX и PC/DOS могли совместно работать в одной и той же распределенной системе.

11. Независимость от сети

Здесь также можно ограничиться кратким замечанием о том, что если система в состоянии поддерживать несколько узлов с разным аппаратным обеспечением и разными операционными системами, то было бы желательно, чтобы в ней поддерживались также разные типы сетей.

12. Независимость от СУБД

Эта цель подразумевает использование несколько менее точной формулировки предположения о строгой однородности. В новой форме это предположение означает, что все экземпляры СУБД на различных узлах поддерживают один и тот же интерфейс, хотя они не обязательно должны быть копиями одного и того же программного обеспечения. Например, если бы системы INGRES и ORACLE поддерживали официальный стандарт языка SQL, то было бы возможно организовать их совместную работу в контексте распределенной системы. Иначе говоря, распределенная система, по крайней мере в некоторой степени, может быть неоднородной.

Поддержка этой неоднородности весьма желательна. Дело в том, что в реальном мире работа обычно организована не только на компьютерах разных типов и в разных операционных системах, но и с участием различных СУБД. Было бы прекрасно, если бы такие СУБД могли работать совместно в одной распределенной системе. Иначе говоря, в идеальной распределенной системе предполагается поддержка независимости от СУБД.

Этой довольно обширной и очень важной с практической точки зрения теме посвящен отдельный раздел.

12.1. Шлюзы

Независимость от СУБД не предполагает строгой однородности. На самом деле в такой ситуации достаточно, чтобы СУБД на разных узлах поддерживали одинаковый интерфейс. Если бы системы INGRES и ORACLE поддерживали официальный стандарт языка SQL, то их, несомненно, можно было бы использовать совместно в одной неоднородной распределенной системе (действительно, такая возможность является одним из аргументов в пользу стандарта языка SQL). Рассмотрим эту возможность подробнее.

Приведенное здесь рассуждение описано на основе систем INGRES и ORACLE всего лишь для того, чтобы конкретизировать стиль изложения, хотя эту идею можно использовать в более общем смысле. Предположим теперь, что существует два узла - *X* и *Y*, на которых запущены соответственно системы INGRES и ORACLE. Допустим также, что некоторый пользователь *U* на узле *X* желает взглянуть на распределенную базу данных, которая содержит данные из базы данных INGRES на узле *X* и из базы данных ORACLE на узле *Y*. По определению *U* является пользователем системы INGRES, а потому работа с распределенной базы данных должна походить на работу базы данных INGRES. Таким образом, режим работы пользователя должен поддерживаться системой INGRES, а не системой ORACLE. В чем заключается такая поддержка?

В принципе, она организована довольно просто. Система INGRES должна иметь *надстроечную* программу, которая называется шлюзом и взаимодействует с системой ORACLE так, чтобы создавалось впечатление, что "работа системы ORACLE выглядит так же, как и работа системы INGRES". Такая распределенная система показана на рис. 2.3.

Тогда шлюз должен выполнять все перечисленные ниже функции, причем некоторые из них с точки зрения технического воплощения представляют собой весьма сложную задачу. (Более подробно эти проблемы обсуждаются ниже при описании стандартов удаленного доступа.)

- Воплощение протоколов обмена информацией между системами INGRES и ORACLE. Такие протоколы, помимо прочего, должны содержать сведения о совместимости формата сообщений, с помощью которых утверждения SQL посылаются со стороны системы INGRES, а также посылаются сведения об отображении результатов в системе ORACLE (значения данных, коды возврата и т.д.) в формат сообщения, которое понимала бы система INGRES.
- Обеспечение функции "реляционного сервера" для системы ORACLE (аналогичной функции, выполняемой интерактивным процессором языка SQL, который уже поддерживается в большинстве современных программных продуктов такого типа). Иначе говоря, шлюз должен обрабатывать произвольные незапланированные утверждения языка SQL, заданные для базы данных ORACLE. Для того чтобы обеспечить эту функцию, шлюз должен осуществлять динамическую поддержку языка SQL или поддерживать интерфейс вызовов на узле системы ORACLE.
- Отображение между типами данных систем INGRES и ORACLE. Эта проблема включает множество подчиненных проблем, которые связаны с различием процессоров (например, у разных процессоров может по-разному определяться длина переменной типа "word"), различием кодировок символов (при сравнении строк в запросах упорядочения могут быть получены неожиданные результаты), различием формата чисел с плавающей запятой (широко известная проблема), различием в способе управления датой и временем (никакие СУБД не поддерживают это управление одинаковым образом) и т.д.
- Отображение диалекта языка SQL, принятого в системе INGRES, на диалект SQL, принятый в ORACLE. Следует отметить, что как в системе INGRES, так и в системе ORACLE поддерживаются отдельные инструменты языка SQL, которые не поддерживаются в другой системе. К тому же в них содержатся некоторые конструкции, синтаксически одинаковые, но семантически различные.
- Отображение информации, посылаемой обратно от системы ORACLE (код возврата и т.д.), в формате INGRES.
- Отображение каталога ORACLE в формате INGRES таким образом, чтобы узел INGRES и его пользователи могли понимать содержимое базы данных ORACLE.
- Работа в качестве участника (в варианте INGRES) протокола двухфазной фикса-

ции (при условии, что транзакции INGRES разрешено выполнение обновлений в базе данных ORACLE). Сможет ли данный шлюз выполнить эту функцию, зависит от модулей, предоставляемых администратором транзакций на узле системы ORACLE. Стоит отметить, что в коммерческих вариантах администраторов транзакций (за некоторым исключением) *не* обеспечивалась необходимая для этого поддержка. Иными словами не предусматривалась способность прикладной программы сообщить администратору транзакций о "подготовке к прекращению выполнения" (как альтернатива сообщению о безусловном прекращении выполнения, т.е. завершении или отмене выполнения).

- Данные на узле системы ORACLE, которые необходимо блокировать по запросу узла системы INGRES, должны блокироваться тогда, когда система INGRES нуждается в этом. Снова следует отметить, что шлюз сможет выполнить эту функцию в зависимости от того, будет ли архитектура блокировки системы ORACLE совпадать с системой INGRES или нет.

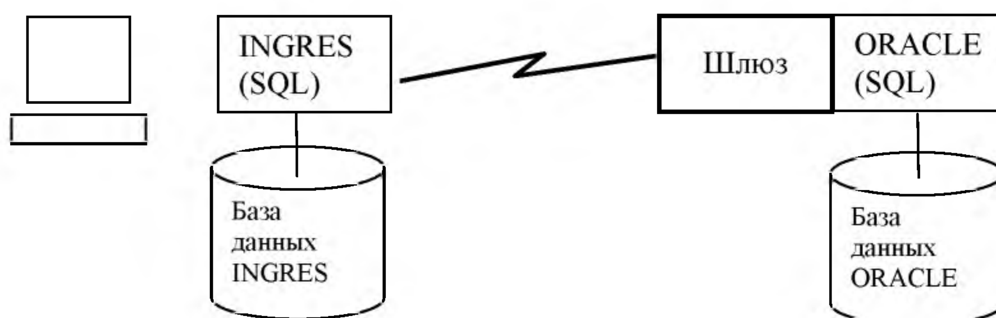


Рис.2.3. Гипотетический шлюз, обеспечиваемый системой INGRES для обмена данными с системой ORACLE

До сих пор независимость от СУБД обсуждалась только в контексте реляционных систем, а что будет, если в реляционную нераспределенную систему включить нереляционный узел. Например, можно ли организовать доступ к узлу системы IMS с узлов систем INGRES или ORACLE? Возможность реализации такого доступа была бы весьма полезной с практической точки зрения, поскольку позволила бы использовать огромное количество данных, хранящихся в системе IMS и других системах "дореляционного периода". Однако как это можно сделать?

Если этот вопрос сформулировать конкретнее, а именно: "можно ли это выполнить на все 100 %", т.е. "можно ли обеспечить доступ ко всем нереляционным данным с помощью реляционного интерфейса и могут ли все реляционные операции быть выполнены с этими данными", то ответ будет определенно *отрицательным*. Но если этот же вопрос сформулировать иначе, а именно: "можно ли в такой системе обеспечить некоторый уровень полезной функциональности", то ответ будет *положительным*.

В заключение следует отметить, что при организации удовлетворительной работы шлюзов часто возникают существенные проблемы, особенно если система вывода данных не является реляционной. Однако потенциальные выгоды весьма велики, даже если методы их воплощения не очень совершенны. Поэтому на рынке программного обеспечения уже появилось много программных продуктов с поддержкой шлюзов, к тому же в ближайшие годы можно ожидать дальнейшего увеличения их количества. Однако следует предупредить, что несмотря на заверения производителей о совершенности предлагаемых ими изделий, как правило, дело обстоит совсем наоборот. Caveat emptor (Покупатель действует на свой страх и риск).

Лекция 3. Проблемы распределенных систем

Ниже более подробно описываются некоторые уже упомянутые проблемы. Основная проблема информационных сетей, по крайней мере глобальных, заключается в том, что они достаточно медленны. В типичной глобальной сети интенсивность обмена данными равна приблизительно 5 или 10 тыс. байт в секунду, а для обычного жесткого диска интенсивность обмена данными - около 5 или 10.млн байт в секунду. (С другой стороны, в локальных вычислительных сетях уже может поддерживаться скорость обмена данными, сравнимая с интенсивностью обмена на жестких дисках.) Вследствие этого основным требованием к распределенным системам является минимизация использования сети, т.е. сокращение до минимума количества и объема пересылаемых в сети сообщений. Стремление к достижению-этой цели приводит, в свою очередь, к необходимости решения перечисленных ниже проблем (хотя перечень всех проблем не исчерпывается этим списком).

- Обработка запросов.
- Управление каталогом.
- Распространение обновления.
- Управление восстановлением.
- Управление параллелизмом.

Обработка запросов

При минимизации использования сети предполагается, что сама по себе оптимизация запроса, как и его исполнение, должна быть распределенной. Иначе говоря, общий процесс оптимизации обычно состоит из этапа глобальной оптимизации, который сопровождается несколькими этапами локальной оптимизации. Например, допустим, что запрос Q задан на узле X и включает объединение отношения R_y, содержащего сто кортежей на узле Y, и отношения R_z, содержащего миллион кортежей на узле Z . Оптимизатор на узле X выберет глобальную стратегию для выполнения запроса Q, при этом, очевидно, лучше переместить отношение R_y на узел Z, а не отношение R_z на узел Y (и конечно же, не следует перемещать оба отношения R_y и R_z на узел X. Тогда сразу после перемещения отношения R_y на узел Z стратегия выполнения объединения на узле Z будет выбрана локальным оптимизатором на узле Z.

Проиллюстрируем изложенные идеи. Рассмотрим *базу данных* поставщиков и деталей (упрощенный вариант):

S	{ S#, CITY }	10 000 хранимых кортежей на узле A
P	{ P#, COLOR }	100 000 хранимых кортежей на узле B
SP	{ S#, P# }	1 000 000 хранимых кортежей на узле A

Предполагается, что каждый хранимый кортеж имеет размер 25 байт (200 бит).

Запрос "Получить сведения о находящихся в Лондоне (London) поставщиках красных (Red) деталей:

```
S.S# WHERE EXISTS SP EXISTS P  ( S.CITY = 'London' AND
                                 S.S# = SP.S# AND
                                 SP.P# = P.P# AND
                                 P.COLOR = 'Red' )
```

Оценочные границы промежуточных результатов:

Число красных деталей	= 10
Число поставок, выполняемых поставщиками из Лондона	= 100 000

Предполагаемые параметры обмена данными в сети:

Интенсивность обмена данными = 50 000 бит/с
Задержка доступа = 0,1 с

Теперь можно вкратце рассмотреть шесть возможных стратегий обработки этого запроса с вычислением для каждой *i*-стратегии общего времени передачи данных Т[*i*] по следующей формуле:

Т[*i*] = общая задержка доступа + (общий объем данных / интенсивность обмена данными) = (число сообщений / 10) +(число бит / 50 000)

1. Переместить отношение *P* на узел *A* и выполнить запрос на узле *A*.

Т[1] = 0,1 + (100 000*200)/ 50 000 = приблизительно 400 с (6,67 мин)

2. Переместить отношения *S* и *SP* на узел *B* и выполнить запрос на узле *B*.

Т[2] = 0,2 + ((10 000 + 1 000 000)*200)/50 000 = приблизительно 4 040 с (1,12 ч)

3. Соединить отношения *S* и *SP* на узле *A*, выбрать из полученного результата кортежи для поставщиков из Лондона, а затем для каждого кортежа на узле *B* проверить, не является ли соответствующая деталь красной. Каждая из этих проверок будет содержать два сообщения: запрос и ответ. Время передачи данных для таких сообщений будет значительно меньше по сравнению с задержкой доступа.

Т[3] = приблизительно 20 000 с (5,56 ч)

4. Выбрать из отношения *P* на узле *B* кортежи, соответствующие красным деталям, а затем для каждого кортежа на узле *A* проверить, не поставляется ли соответствующая деталь поставщиком из Лондона. Каждая из этих проверок будет содержать два сообщения: запрос и ответ. Время передачи данных для этих сообщений опять будет значительно меньше по сравнению с задержкой доступа.

Т[4] = приблизительно 2 с

5. Соединить отношения *S* и *SP* на узле *A*, выбрать из полученного результата кортежи для поставщиков из Лондона, результат разбить на проекции по атрибутам *S#* и *P#*, а затем переместить на узел *B*. Завершить выполнение запроса на узле В.

Т[5] = 0,1 + (100 000*200)/50 000 = приблизительно 400 с (6,67 ч)

6. Выбрать из отношения *P* на узле *B* кортежи, соответствующие красным деталям, а затем переместить результат на узел А. Завершить выполнение запроса на узле А.

Т[6] = 0,1 + (10*200)/50 000 = приблизительно 0,1 с

На рис. 3.1 эти результаты представлены в одной таблице

Стратегия	Метод	Время передачи данных
1	Переместить отношение *P* на узел *A*	
2	Переместить отношение *S* и *SP* на узел *B*	1,12 ч
3	Для каждой поставки из Лондона проверить, является ли деталь красной	5,56 ч
4	Для каждой ли красной детали проверить, не поставляется ли она из Лондона	2 с
5	Переместить сведения о поставках из Лондона на узел *B*	6,67 мин
6	Переместить сведения о красных деталей	0,1 с (наилучший результат)

Рис. 3.1. Стратегии распределенного выполнения запроса (итоги)

Внимательно ознакомившись с этими результатами, можно отметить следующие

важные особенности:

1. Каждая из шести стратегий представляет собой один из возможных подходов к решению этой проблемы, несмотря на очень значительные вариации во времени передачи данных.
2. Интенсивность обмена данными и задержка доступа являются важными факторами, влияющими на выбор той или иной стратегии.
3. Для плохих стратегий продолжительность операций вычисления и ввода-вывода данных пренебрежимо мала по сравнению со временем передачи данных. (Это в большей или меньшей мере может относиться и к более совершенным стратегиям.)

В дополнение к сказанному выше следует отметить, что некоторые стратегии позволяют выполнять параллельную обработку на двух узлах. Таким образом, время отклика в такой системе может оказаться меньше времени отклика в централизованной системе. Обратите внимание, что в данном обсуждении игнорировался вопрос о том, какой узел получает окончательные результаты.

Управление каталогом

Каталог распределенной системы содержит не только обычные данные, касающиеся базовых отношений, представлений, индексов, пользователей и т.д., но также и всю информацию, необходимую для обеспечения независимости размещения, фрагментации и репликации. В таком случае возникает вопрос: где и как следует хранить системный каталог? Ниже перечислены некоторые варианты хранения системного каталога.

1. *Централизованный каталог*. Весь каталог хранится в одном месте, т.е. на центральном узле.
2. *Полностью реплицированный каталог*. Весь каталог полностью хранится на каждом узле.
3. *Секционированный каталог*. На каждом узле содержится его собственный каталог для объектов, хранимых на этом узле. Общий каталог является объединением всех разъединенных локальных каталогов.
4. *Комбинация первого и третьего вариантов*. На каждом узле хранится собственный локальный каталог (как в п. 3), кроме того, на одном центральном узле хранится унифицированная копия всех этих локальных каталогов (как в п. 1).

Для каждого подхода характерны определенные недостатки и проблемы. В первом подходе, очевидно, не достигается "независимость от центрального узла". Во втором утрачивается автономность функционирования, поскольку при обновлении каждого каталога это обновление придется распространить на каждый узел. В третьем выполнение нелокальных операций становится весьма дорогостоящим (для поиска удаленного объекта потребуется в среднем осуществить доступ к половине имеющихся узлов). Четвертый подход более эффективен, чем третий (для поиска удаленного объекта потребуется осуществить доступ только к одному удаленному каталогу), но в нем снова не достигается "независимость от центрального узла". В традиционных системах могут использовать *другие* подходы. В качестве примера здесь рассматривается подход, использованный в системе R*.

Для того чтобы описать структуру каталога системы R*, необходимо прежде всего рассмотреть принятый в ней порядок **именования объектов**. Именование объектов является существенным аспектом распределенных систем, поскольку, если два разных узла *A* и *B* содержат хранимое отношение под одинаковым именем *R*, это приводит к необходимости выработки некоторого метода обеспечения уникальности имен в рамках всей системы. Однако при указании пользователю уточненного имени (например, *A.R* и *B.R*) нарушается требование независимости расположения. В таком случае необходимо отображение известных пользователям имен на соответствующие системные имена.

В системе R* используется следующий подход к этой проблеме. В ней вводятся понятия **печатного имени**, т.е. имени объекта, на которое обычно ссылаются пользо-

ватели (например, в выражениях SQL), а также системного имени, которое является глобальным уникальным внутренним идентификатором этого объекта. Системное имя содержит четыре компонента.

- *Идентификатор создателя* объекта.
- *Идентификатор узла создателя*, т.е. узла, на котором была выполнена операция создания объекта.
- *Локальное имя*, т.е. неуточненное имя объекта.
- *Идентификатор узла хранения*, т.е. узла, на котором этот объект хранился в исходном состоянии.

Так, системное имя

MARYLIN @ NEW YORK . STATS @ LONDON

идентифицирует объект (например, хранимое отношение) с локальным именем STATS, созданный пользователем Marylin на узле в Нью-Йорке и изначально хранившийся на узле в Лондоне. Такое имя **гарантировано от каких-либо изменений** даже при перемещении этого объекта на другой узел (подробнее это объясняется ниже).

Как уже отмечалось, пользователи обычно применяют к объектам их *печатные имена*, которые состоят из простого неуточненного имени: например, либо "локального компонента" системного имени (STATS в рассматриваемом примере), либо **синонима** системного имени, определенного с помощью специальной инструкции CREATE SYNONYM языка SQL, используемого в системе R*. Ниже приводится пример такой инструкции.

CREATE SYNONYM MSTATS FOR MARYLIN @ NEW YORK . STATS @ LONDON;

Теперь можно использовать одно из следующих выражений:

SELECT . . . FROM STATS . . . ;
SELECT . . . FROM MSTATS . . . ;

В первом случае (при использовании локального имени) системное имя может быть выведено на основе очевидных и принятых по умолчанию предположений, а именно на основе того, что данный объект был создан данным пользователем на данном узле и изначально хранился на этом узле. Одним из следствий такого способа будет то, что старые приложения для System R могут быть без всяких изменений запущены на выполнение в системе R* (т.е. сразу после переопределения данных для системы R*).

Во втором случае (при использовании синонимов) системное имя определяется помощью опроса соответствующей **таблицы синонимов**. Эти таблицы рассматриваются как первый компонент каталога, а каждый узел содержит набор таблиц всех пользователей, известных на данном узле, с отображением синонимов пользователя на системные имена.

В дополнение к таблицам синонимов на каждом узле поддерживаются следующие объекты:

1. Элемент каталога для каждого объекта, **созданного** на этом узле.
2. Элемент каталога для каждого объекта, **хранимого** в данный момент на этом узле.

Допустим, пользователь создает запрос для поиска синонима MSTATS. Сначала система ищет соответствующее ему системное имя в соответствующей таблице синонимов (чисто локальная подстановка). После этого становится известным место создания объекта; в рассматриваемом примере это Лондон. Затем система опрашивает каталог Лондона, что в общем случае приводит к подстановке с удаленным доступом (первый удаленный доступ). Каталог Лондона будет содержать элемент для данного объекта согласно упомянутому выше пункту 1. Если искомый объект находится все еще в

Лондоне, то он будет немедленно найден. Однако, если он перемещен, например в Лос-Анджелес, в таком случае это будет указано в элементе каталога Лондона. Благодаря такой организации система теперь может опросить каталог Лос-Анджелеса (второй удаленный доступ), который согласно пункту 2 будет содержать элемент для искомого объекта. Таким образом, этот объект будет найден, по крайней мере, с помощью двух попыток удаленного доступа.

Более того, при очередной миграции объекта, например в Сан-Франциско, в системе будут выполнены следующие действия:

- Вставлен элемент каталога Сан-Франциско.
- Удален элемент каталога Лос-Анджелеса.
- Элемент каталога Лондона, указывающий на Лос-Анджелес, будет заменен элементом каталога, указывающим на Сан-Франциско.

Общий эффект от их выполнения заключается в том, что объект снова может быть найден с помощью всего лишь двух попыток удаленного доступа. К тому же такая система является действительно распределенной, так как в ней нет каталога на центральном узле, а также не существует никакой другой возможности для глобального краха системы.

Следует отметить, что в модуле распределенной работы в системе DB2 используется схема именования объектов, похожая на описанную выше, но не идентичная ей.

Распространение обновления

Как указывалось выше, основной проблемой репликации данных является то, что обновление любого логического объекта должно распространяться на все хранимые копии этого объекта. Трудности возникают из-за того, что некоторый узел, содержащий данный объект, может быть недоступен (например, из-за краха системы или данного узла) именно в момент обновления. В таком случае очевидная стратегия немедленного распространения обновлений на все копии может оказаться неприемлемой, поскольку предполагается, что обновление (а значит, и исполнение транзакции) будет провалено, если одна из этих копий будет недоступна в текущий момент. В некотором смысле, при использовании такой стратегии данные действительно будут менее доступны, чем при их использовании в нереплицированном виде. Таким образом, существенно подрывается одно из преимуществ репликации, упомянутое в предыдущем разделе.

Общая схема устранения этой проблемы (и не единственно возможная в этом случае), называемая схемой **первичной копии**, описана ниже.

- Одна копия каждого реплицируемого объекта называется *первичной* копией, а все остальные - *вторичными*.
- Первичные копии различных объектов находятся на различных узлах (таким образом, эта схема является распределенной).
- Операции обновления считаются завершенными, если обновлены все первичные копии. В таком случае *в некоторый момент* времени узел, содержащий такую копию, несет ответственность за распространение операции обновления на вторичные копии. Однако поскольку свойства транзакции АСИД (атомарность, согласованность, изоляция, долговечность) должны выполняться, то подразумевается, что "некоторый момент" времени предшествует завершению исполнения транзакции.

Конечно, данная схема приводит к некоторым дополнительным проблемам. Обратите внимание, что эти проблемы приводят к нарушению требования локальной автономии, поскольку даже если локальная копия остается доступной, выполнение транзакции может потерпеть неудачу из-за недоступности удаленной (первичной) копии некоторого объекта.

Под требованием атомарности транзакции подразумевается, что распространение всех операций обновления должно быть закончено до завершения соответствующей

транзакции. Однако существовало несколько коммерческих программных продуктов с поддержкой менее амбициозной формы репликации. В ней распространение обновления *гарантировалось* в будущем (вероятно, в некоторое заданное пользователем время), но не обязательно в рамках соответствующей транзакции. К сожалению, термин "репликация" в некоторых программных продуктах был использован в несколько ином смысле, чем следует. В результате на рынке программного обеспечения утвердилось мнение, что распространение обновления откладывается вплоть до завершения соответствующей транзакции. Проблема такого "откладываемого распространения" заключается в том, что пользователь в некоторый заданный момент времени не знает, согласована база данных или нет. Поэтому в базе данных не может быть гарантирована совместимость в произвольный момент времени.

В заключение стоит привести несколько дополнительных замечаний в отношении откладываемого распространения обновления.

1. Концепция репликации в системе с откладываемым распространением обновления может рассматриваться как ограниченное воплощение идеи снимков.
2. Одна из причин (возможно, самая главная) использования в программных продуктах откладываемого распространения обновления заключается в том, что для обновления всех реплик до завершения транзакции требуется поддержка протокола двухфазной фиксации, для которого, в свою очередь, требуется исправность всех соответствующих узлов и их готовность к запуску во время выполнения транзакции, что отрицательно влияет на производительность. Такое положение дел характеризуется появлением в печати статей с загадочными заголовками типа "Репликация или двухфазная фиксация". Причем их загадочность вызвана тем, что сравниваются преимущества двух совершенно разных подходов.

Управление восстановлением

Управление восстановлением в распределенных системах обычно основано на протоколе **двухфазной фиксации** (или некотором варианте этого протокола). Поддержка двухфазной фиксации необходима в *любой* среде, в которой одна транзакция может взаимодействовать с несколькими автономными администраторами ресурсов. Однако она особенно важна в распределенной системе, поскольку администраторы ресурсов, т.е. локальные СУБД, действуют на разных узлах и, следовательно, автономны.

Здесь необходимо отметить несколько важных особенностей.

1. Стремление к "независимости от центрального узла" означает, что функция координатора не должна присваиваться ни одному из узлов сети, вместо этого она должна выполняться для разных транзакций различными узлами. Обычно она выполняется узлом, на котором данная транзакция была запущена. Таким образом, каждый узел должен для одних транзакций выполнять роль узла-координатора, а для других - узла-участника.
2. При двухфазной фиксации требуется, чтобы координатор обменивался данными с каждым узлом-участником, что, в свою очередь, означает большее количество сообщений и больше накладных расходов.
3. Если узел *Y* действует как участник процесса двухфазной фиксации, координируемого узлом *X*, то узел *Y* должен выполнять любые действия, предписываемые узлом *X* (например, завершение или отмену выполнения транзакции). При этом неизбежна утрата локальной автономности.
4. В идеальной ситуации, конечно, хотелось бы, чтобы процесс двухфазной фиксации продолжался несмотря на неисправность сети или отдельных узлов либо был устойчив по отношению к любым возможным видам неисправности. К сожалению, легко заметить, что эта проблема неразрешима в принципе, т.е. не существует никакого конечного протокола, который мог бы гарантировать, что все агенты одновременно завершат выполнение успешной транзакции либо отменят выпол-

нение неуспешной транзакции несмотря на неисправности произвольного характера. Можно предположить и обратную ситуацию, т.е. что такой протокол все-таки существует. Пусть минимальное количество сообщений, необходимых для осуществления этого протокола, равняется *N*. Предположим теперь, что последнее из этих *N* сообщений утеряно по какой-то причине. Тогда либо это сообщение не является необходимым, что противоречит исходному предположению о минимально необходимом количестве сообщений *N*, либо такой протокол не будет работать. Любой из этих случаев ведет к противоречию, из чего можно заключить, что такого протокола не существует.

5. Протокол двухфазной фиксации может рассматриваться как *основной* протокол. Однако существует множество модификаций этого основного протокола, которые можно и следовало бы применить на практике. Например, в варианте под названием протокол *предполагаемой фиксации*, в котором при успешном выполнении транзакции число сообщений уменьшается за счет введения дополнительных сообщений для случая неуспешного выполнения транзакции.

Управление параллелизмом

Управление параллелизмом в большинстве распределенных систем, как и во многих нераспределенных системах, основано на блокировке. Однако в распределенной системе запросы на проверку, установку и снятие блокировок являются *сообщениями* (здесь предполагается, что рассматриваемый объект находится на удаленном узле), что влечет за собой дополнительные накладные расходы. Рассмотрим, например, транзакцию *T*, которая нуждается в обновлении объекта, имеющего реплики на *n* удаленных узлах. Если каждый узел управляет блокировками для объектов, хранимых на этом узле (как это было бы при соблюдении локальной автономии), то для простейшего способа управления параллелизмом потребовалось бы по крайней мере 5*n* сообщений:

n запросов на блокировку,
n разрешений на блокировку,
n сообщений об обновлении,
n подтверждений,
n запросов на снятие блокировки.

Конечно, можно легко усовершенствовать этот способ, используя подтверждения, вложенные в блок данных обратного направления. Таким образом могут комбинироваться запрос на блокировку и сообщения об обновлении, а также разрешение на блокировку и подтверждения. Но даже в таком случае общее время обновления может быть на несколько порядков выше, чем в централизованной системе.

Обычный подход к этой проблеме заключается в принятии стратегии *первичной копии*, которая в краткой форме была представлена выше. Для данного объекта *R* узел, содержащий первичную копию объекта *R*, будет управлять всеми операциями блокировки, включающими *R* (помните, что первичные копии разных объектов в общем случае могут быть расположены на различных узлах). При такой стратегии множество всех копий объекта в целях блокировки может рассматриваться как единый объект, а общее число сообщений будет снижено от 5*n* до 2*n*+3 (один запрос на блокировку, одно разрешение на блокировку, *n* обновлений, *n* подтверждений и один запрос на снятие блокировки). Однако при таком решении опять утрачивается автономность, т.е. транзакция может оказаться неуспешной, если первичная копия недоступна (даже при использовании транзакции только для чтения), а локальная копия доступна. Заметьте, что для блокировки первичной копии необходимы не только операции обновления, но также и операции извлечения. Таким образом, неприятный побочный эффект при использовании стратегии первичной копии заключается в снижении производительности и доступности данных как для операций извлечения, так и для операций обновления.

Другой проблемой блокировки в распределенной системе является то, что она может

привести к **глобальному тупику**, который охватывает два или более узлов. На рис. 3.2 представлен пример возникновения такого тупика:

1. Агент транзакции T_2 на узле X ожидает, когда агент транзакции T_1 снимет блокировку на узле X
2. Агент транзакции T_1 на узле X ожидает, когда закончится выполнение транзакции T_1 на узле Y
3. Агент транзакции T_1 на узле Y ожидает, когда агент транзакции T_2 снимет блокировку на узле Y.
4. Агент транзакции T_2 на узле Y ожидает, когда закончится выполнение транзакции T_2 на узле X. Тупиковая ситуация!

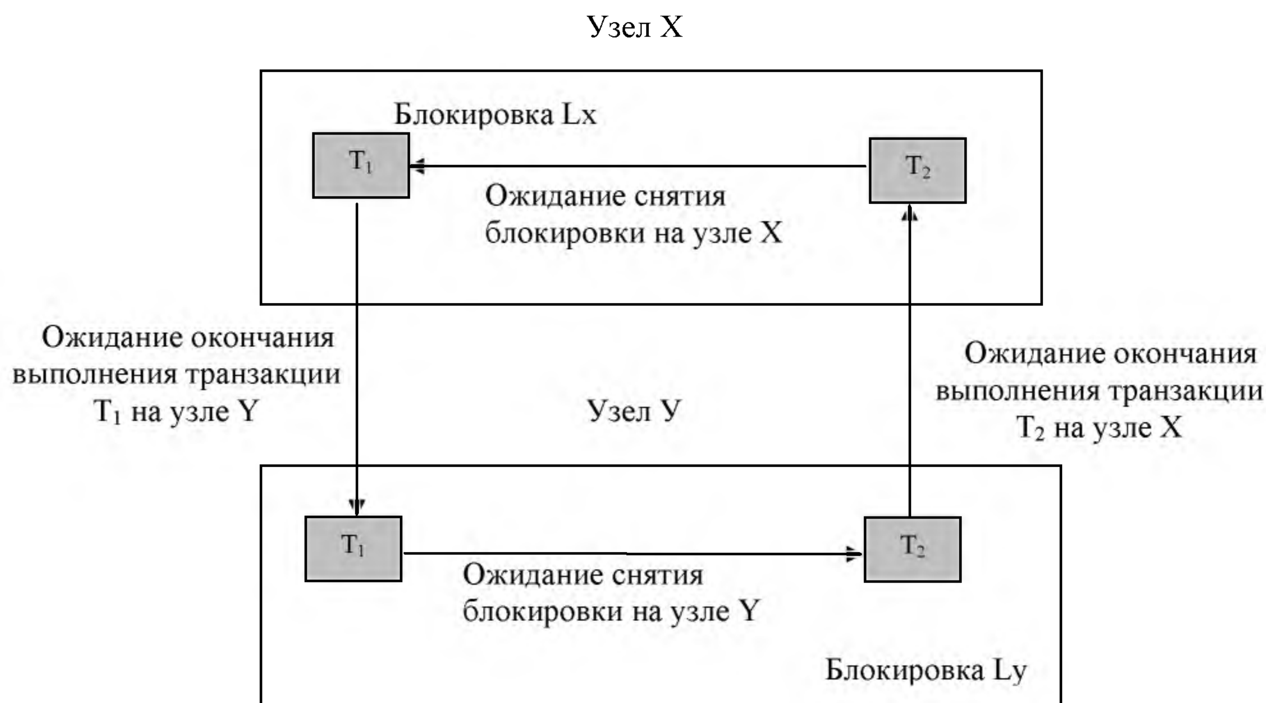


Рис. 3.2. Пример глобального тупика.

Проблема тупика такого типа состоит в том, что *ни один из узлов не может обнаружить тупик, используя только информацию, которая сосредоточена на этом узле*. Иначе говоря, в локальных диаграммах ожидания нет никаких циклов, но они появятся при объединении локальных диаграмм в глобальную диаграмму ожидания. Отсюда следует, что обнаружение глобальных тупиков связано с увеличением накладных расходов, поскольку для этого требуется дополнительно совместить отдельные локальные диаграммы.

Весьма элегантная (распределенная) схема обнаружения глобального тупика приводится в публикациях, посвященных системе R*.

Лекция 4. Функция восстановления

Функции восстановления и параллелизма во многом взаимодействуют и являются аспектами более общей проблемы, называемой ***процессом транзакции***. Однако из педагогических соображений желательно изучать их отдельно. В настоящей лекции остановимся на функции восстановления, однако время от времени ссылки на функцию параллелизма будут неизбежны.

Восстановление в системе управления базами данных означает в первую очередь восстановление самой базы данных, т.е. возвращение базы данных в правильное состояние, если какой-либо сбой сделал текущее состояние неправильным или подозрительным.

Основной принцип, на котором строится такое восстановление, достаточно прост - это избыточность. Но эта избыточность организуется на физическом уровне. Конечно, такая избыточность будет скрыта от пользователя, а следовательно, не видна на логическом уровне. Другими словами, если любая часть информации, содержащаяся в базе данных, может быть реконструирована из другой хранимой в системе избыточной информации, значит, база данных восстанавливаема.

Перед тем как идти дальше, необходимо уяснить, что принцип восстановления и процесс транзакции в целом не зависят от того, является ли базовая система реляционной или какой-либо еще. Нужно также отметить, что это весьма обширный предмет обсуждения, и мы познакомимся только с наиболее важными и базовыми принципами.

Транзакции

Транзакция - это логическая единица работы. Рассмотрим пример. Предположим сначала, что отношение P (отношение деталей) включает дополнительный атрибут TOTQTY, представляющий собой общий объем поставок для каждой детали. Значение TOTQTY для любой определенной детали предполагается равным сумме всех значений QTY для всех поставок данной детали. На рис. 4.1 показано добавление в базу данных новой поставки со значением 1000 для поставщика S_5 и детали P_1 (INSERT добавляет новую поставку к отношению SP, а UPDATE обновляет значение TOTQTY для детали P_1).

```
BEGIN TRANSACTION;
        INSERT ( { S# : ' S5 ', P# : ' P1 ' , QTY : 1000 } )    INTO SP ;
        IF возникла ошибка THEN GO TO UNDO ;
        UPDATE P WHERE P# = ' P1'  TOTQTY : = TOTQTY + 1000 ;
        IF возникла ошибка THEN GO TO UNDO ;
        COMMIT TRANSACTION;
        GO TO FINISH ;
UNDO :
        ROLLBACK TRANSACTION;
FINISH:
        RETURN ;
```

Рис. 4.1. Пример транзакции (псевдокод)

В приведенном примере предполагается, что речь идет об одиночной, атомарной операции. На самом деле добавление новой поставки - это выполнение двух обновлений в базе данных (под обновлениями здесь, конечно, понимаются операции INSERT, DELETE, а также сами по себе операции UPDATE). Более того, в базе данных между этими двумя обновлениями временно нарушается требование, что значение TOTQTY для детали P_1 равно сумме всех значений QTY для этой детали. Таким образом, логическая единица работы (т.е. транзакция) - не просто одиночная операция системы баз данных, а скорее

согласование нескольких таких операций. В общем, это преобразование одного согласованного состояния базы данных в другое, причем в промежуточных точках база данных находится в несогласованном состоянии.

Из этого следует, что не допустимо, чтобы одно из обновлений было выполнено, а другое нет, так как база данных останется в несогласованном состоянии. В идеальном случае должны быть выполнены оба обновления. Однако нельзя обеспечить стопроцентную гарантию, что так и будет. Система, поддерживающая процесс транзакции, обеспечивает гарантию, что если во время выполнения неких обновлений произошла ошибка (по любой причине), то все эти обновления будут аннулированы. Таким образом, транзакция или выполняется полностью, или полностью отменяется (как будто она вообще не выполнялась).

Системный компонент, обеспечивающий атомарность (или ее подобие), называется ***администратором транзакций*** (или диспетчером транзакций), а ключами к его выполнению служат операторы COMMIT TRANSACTION и ROLLBACK TRANSACTION.

- Оператор COMMIT TRANSACTION (для краткости COMMIT) сигнализирует об успешном окончании транзакции. Он сообщает администратору транзакций, что логическая единица работы завершена успешно, база данных вновь находится (или будет находиться) в согласованном состоянии, а все обновления, выполненные логической единицей работы, теперь могут быть зафиксированы, т.е. стать постоянными.
- Оператор ROLLBACK TRANSACTION (для краткости ROLLBACK) сигнализирует о неудачном окончании транзакции. Он сообщает администратору транзакций, что произошла какая-то ошибка, база данных находится в несогласованном состоянии и все обновления могут быть отменены, т.е. аннулированы.

Следовательно, COMMIT выполняется, если оба обновления прошли успешно, изменения в базе данных выполнены и стали постоянными. Если что-то не так, если обновление прервано каким-либо условием ошибки, то выполняется ROLLBACK и любые изменения отменяются.

Обратите внимание, что при этом можно не только модифицировать базу данных (или попытаться это сделать), но также и выдать пользователю некоторое сообщение о том, что произошло. Например, можно выдать сообщение "Поставка добавлена", если выполнена операция COMMIT, или "Ошибка - поставка не добавлена", если операция не выполнена. Выдача сообщений имеет дополнительное значение для восстановления.

Система поддерживает *файл регистрации*, или журнал, на ленте или (обычно) на диске, где записываются детали всех операций обновления, в частности новое и старое значения модифицированного объекта. Таким образом, при необходимости отмены некоторого обновления система может использовать соответствующий файл регистрации для возвращения объекта в первоначальное состояние. На самом деле это слишком упрощенное объяснение. На практике файл регистрации состоит из двух частей:

- активной (или интерактивной);
- архивной (или автономной).

Интерактивная часть используется во время нормальной системной операции по записи деталей осуществляемых обновлений и обычно содержится на диске. После того как интерактивная часть заполнена, ее содержимое перемещается в автономную часть, которая обычно находится на ленте, поскольку обрабатывается последовательно.

Еще один важный момент. Система должна гарантировать, что индивидуальные операторы сами по себе атомарны (т.е. выполняются полностью или не выполняются совсем). Это особенно важно для реляционных систем, в которых операторы многоуровневые и обычно оперируют множеством кортежей одновременно. Такой оператор просто не может быть нарушен посреди операции и привести систему в несогласованное

состояние (например, если выполнена только часть необходимых действий). Другими словами, если произошла ошибка во время работы такого оператора, база данных должна остаться полностью неизмененной. Более того, это должно быть справедливо даже в том случае, когда действия оператора являются причиной дополнительной, например каскадной, операции.

Восстановление транзакции

Транзакция начинается с успешного выполнения оператора BEGIN TRANSACTION (для краткости BEGIN) и заканчивается успешным выполнением либо оператора COMMIT, либо ROLLBACK. Оператор COMMIT устанавливает так называемую ***точку фиксации*** (которая в коммерческих продуктах также называется ***точкой синхронизации*** (syncpoint)). Точка фиксации соответствует концу логической единицы работы и, следовательно, точке, в которой база данных находится (или будет находиться) в состоянии согласованности. В данном случае термин "база данных" означает доступную для транзакции часть базы данных. Другие транзакции могут выполняться параллельно этой транзакции и создавать изменения в собственных частях базы данных. Таким образом, вся база данных может и не быть в состоянии согласованности в данный момент. Однако в нашем обсуждении игнорируется возможность конкуренции между транзакциями (поскольку такое упрощение существенно не влияет на рассматриваемый вопрос). Выполнение оператора ROLLBACK вновь возвращает базу данных в состояние, в котором она была во время операции BEGIN TRANSACTION, т.е. в предыдущую точку фиксации. Понятие "предыдущая точка фиксации" достаточно корректно даже в случае первой транзакции в программе при условии, что первый оператор BEGIN по умолчанию устанавливает в программе первичную точку фиксации. Ниже перечислены случаи установки точки фиксации:

1. Все обновления, совершенные программой с тех пор, как установлена предыдущая точка фиксации, выполнены, т.е. ***стали постоянными***. Во время выполнения все такие обновления могут расцениваться *только как пробные* (в том смысле, что они могут быть не выполнены, например прокручены назад). Гарантируется, что *однажды зафиксированное обновление так и останется зафиксированным* (это и есть определение понятия "зафиксировано").
2. Все позиционирование базы данных утеряно, и все блокировки кортежей реализованы. Позиционирование базы данных здесь означает, что в любое конкретное время программа обычно адресована определенным кортежам. Эта адресуемость в точке фиксации теряется.

Пункт 2, исключая возможность сохранения некоторой адресуемости и, следовательно, соответствующей блокировки кортежей, также применим, когда транзакция завершена оператором ROLLBACK, а не COMMIT. К пункту 1 это, конечно, не относится. Обратите внимание, что COMMIT и ROLLBACK завершают транзакцию, а не программу. В общем, выполнение программы будет состоять из согласования нескольких транзакций, запущенных одна за другой, как показано на рис. 4.2.

А сейчас давайте вернемся к примеру предыдущего раздела (см. рис. 4.1). Здесь используется явный тест для определения ошибки и выполняется ROLLBACK, если обнаружена любая ошибка. Однако на самом деле программы не всегда могут использовать явные тесты для всех возможных ошибок. Следовательно, система будет выполнять ROLLBACK неявно для любой программы, которая по какой-либо причине не достигла запланированного завершения операций (на рис. 4.1. "запланированное завершение" означает операцию RETURN, которая завершает программу и передает управление назад в систему).

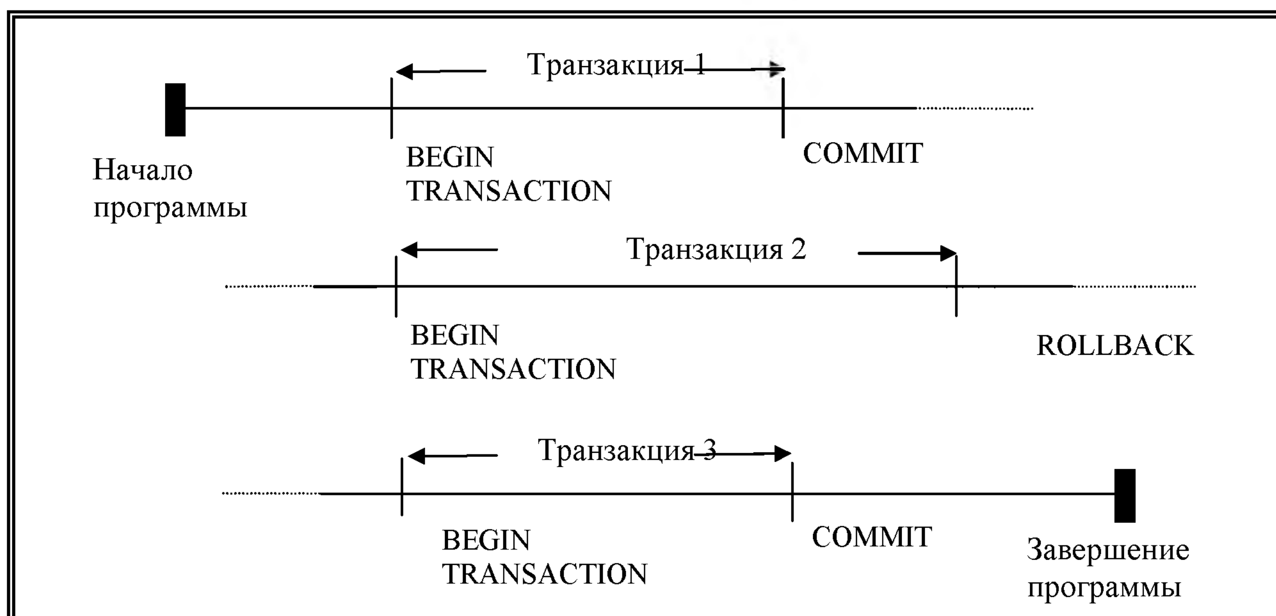


Рис. 4.2. Выполнение программы - это согласование транзакций

Таким образом, *транзакции* - это *не только логические единицы работы*, но также и *единицы восстановления при неудачном выполнении операций*. При успешном завершении транзакции система гарантирует, что обновления постоянно установлены в базе данных, даже если система потерпит крах в следующий момент. Возможно, что в системе произойдет сбой после успешного выполнения COMMIT, но перед тем, как обновления будут физически записаны в базу данных (они все еще могут оставаться в буфере оперативной памяти и таким образом могут быть утеряны в момент сбоя системы). Даже если подобное случилось, процедура перезагрузки системы все равно должна устанавливать эти обновления в базу данных, исследуя соответствующие записи в файле регистрации. Из этого следует, что ***файл регистрации должен быть физически записан перед завершением операции COMMIT***. Это важное правило ведения файла регистрации известно как *протокол предварительной записи в журнал* (т.е. запись об операции осуществляется перед ее выполнением). Таким образом, процедура перезагрузки сможет восстановить любые успешно завершенные транзакции, хотя их обновления не были записаны физически до аварийного отказа системы. Следовательно, как указывалось ранее, транзакция действительно является единицей восстановления.

Согласно предыдущим разделам транзакции обладают четырьмя важными свойствами: *атомарность, согласованность, изоляция* и *долговечность* (они называются ***свойствами АСИД***).

- ***Атомарность***. Транзакции атомарны (выполняется все или ничего).
- ***Согласованность***. Транзакции защищают базу данных согласованно. Это означает, что транзакции переводят одно согласованное состояние базы данных в другое без обязательной поддержки согласованности во всех промежуточных точках. Согласованность базы данных будет детально обсуждаться в следующих главах этой книги.
- ***Изоляция***. Транзакции отделены одна от другой. Это означает, что, если даже будет запущено множество конкурирующих друг с другом транзакций, любое обновление определенной транзакции будет скрыто от остальных до тех пор, пока эта транзакция выполняется. Другими словами, для любых двух отдаленных транзакций T_1 и T_2 справедливо следующее утверждение: T_1 сможет увидеть обновление T_2 только после выполнения T_2, а T_2 сможет увидеть обновление T_1 только после выполнения T_1 .

- ***Долговечность***. Когда транзакция выполнена, ее обновления сохраняются, даже если в следующий момент произойдет сбой системы.

Восстановление системы

Система должна быть готова к восстановлению не только после небольших локальных нарушений, таких как невыполнение операции в пределах определенной транзакции, но также и после глобальных нарушений типа сбоев в питании ЦПУ (центрального процессорного устройства). Местное нарушение по определению поражает только транзакцию, в которой оно собственно и произошло. Такие нарушения были описаны выше в этой главе. Глобальное нарушение поражает сразу все транзакции и, следовательно, приводит к значительным для системы последствиям. Далее мы кратко обсудим, что вовлечено в процесс восстановления при глобальном нарушении в системе. Существует два вида глобальных нарушений.

- ***Отказы системы*** (например, сбои в питании), поражающие все выполняющиеся в данный момент транзакции, но физически не нарушающие базу данных в целом. Такие нарушения в системе также называют аварийным отказом программного обеспечения.
- ***Отказы носителей*** (например, поломка головок дискового накопителя), которые могут представлять угрозу для базы данных или для какой-либо ее части и поражать, по крайней мере, те транзакции, которые используют эту часть базы данных. Отказы носителей также называют аварийным отказом аппаратуры.

Рассмотрим **отказы системы.**

Критической точкой в отказе системы является потеря *содержимого основной (оперативной) памяти* (в частности, рабочих буферов базы данных). Поскольку точное состояние какой-либо выполняющейся в момент нарушения транзакции не известно, транзакция может не завершиться успешно и, таким образом, будет отменена ("прокручена" назад) при перезагрузке системы.

Более того, возможно, потребуется *повторно выполнить* определенную успешно завершившуюся до аварийного отказа транзакцию при перезагрузке системы, если не были физически выполнены обновления этой транзакции.

Возникает вопрос: *как во время перезагрузки система узнает, какую транзакцию отменить, а какую выполнить повторно*? Ответ следующий. В некотором предписанном интервале (когда в журнале накапливается определенное число записей) система автоматически ***принимает контрольную точку***. Принятие контрольной точки включает физическую запись содержимого рабочих буферов базы данных непосредственно в базу данных и специальную физическую запись контрольной точки, которая предоставляет список всех осуществляемых в данный момент транзакций. На рис. 4.3 рассматривается пять возможных вариантов выполнения транзакций до аварийного сбоя системы.

Пояснения к рисунку:

- Отказ системы произошел в момент времени t_f.
- Близлежащая к моменту времени t_f контрольная точка была принята в момент времени t_c.
- Транзакция T_1 успешно завершена до момента времени t_c.
- Транзакция T_2 начата до момента времени t_c и успешно завершена после момента времени t_c, но до момента времени t_f.
- Транзакция T_3 также начата до момента времени t_c, но не завершена к моменту времени t_f.
- Транзакция T начата после момента времени t_c и успешно завершена до момента времени t_f.
- И наконец, транзакция T_5 также начата после момента времени t_c, но не завершена к моменту времени t_f.

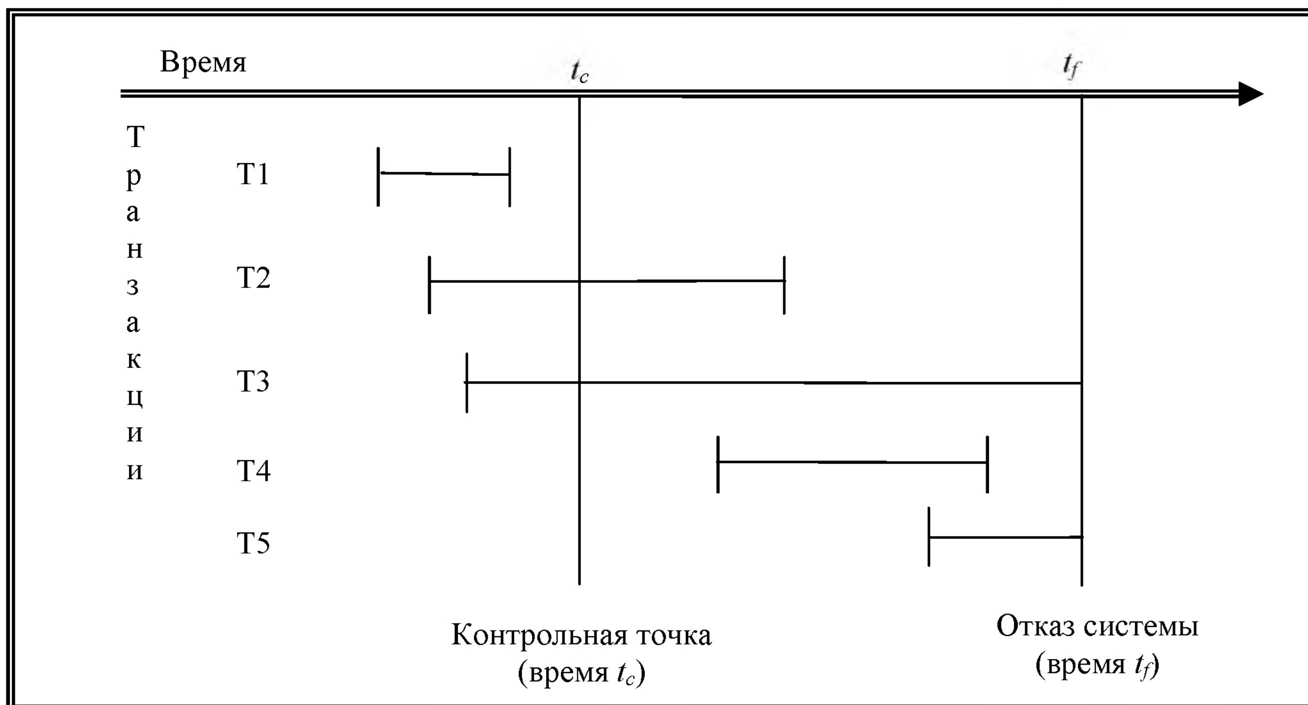


Рис. 4.3. Пять вариантов транзакции

Очевидно, что при перезагрузке системы транзакции типа T_3 и T_5 должны быть отменены, а транзакции типа T_2 и T_4 - выполнены повторно. Тем не менее заметьте, что транзакции типа T_1 вообще не включаются в процесс перезагрузки, так как обновления попали в базу данных еще до момента времени t_c (т.е. зафиксированы еще до принятия контрольной точки). Отметьте также, что транзакции, завершившиеся неудачно (в том числе отмененные) перед моментом времени *tc*, вообще не будут вовлечены в процесс перезагрузки (подумайте, почему).

Следовательно, во время перезагрузки система вначале проходит через процедуру идентификации всех транзакций типа T_2-T_5. При этом выполняются перечисленные ниже шаги.

1. Создается два списка транзакций: назовем их UNDO (отменить) и REDO (выполнить повторно). В список UNDO заносятся все транзакции, предоставленные из записи контрольной точки, т.е. все транзакции, выполняющиеся в момент времени t_c, а список REDO остается пустым.
2. Осуществляется поиск в файле регистрации (журнале), начиная с записи контрольной точки.
3. Если в файле регистрации обнаружена запись BEGIN TRANSACTION о начале транзакции T, то эта транзакция также добавляется в список UNDO.
4. Если в файле регистрации обнаружена запись COMMIT об окончании транзакции T, то эта транзакция добавляется в список REDO.
5. Когда достигается конец файла регистрации, списки UNDO и REDO анализируются для идентификации транзакций типа T_2 и T_4, появившихся в списке REDO, и транзакций типа T_3 и T_5, оставшихся в списке UNDO.

После этого система просматривает назад файл регистрации, отменяя транзакции из списка UNDO, а затем просматривает снова вперед, повторно выполняя транзакции из списка REDO.

Замечание. Восстановление базы данных в правильное состояние путем отмены выполненных операций иногда называется *обратным восстановлением*. Аналогично, восстановление ее в правильное состояние повторным выполнением называется *прямым восстановлением*.

И наконец, когда такая восстановительная работа завершена, система готова при-

ступить к дальнейшей работе.

Восстановление носителей

Процесс восстановления носителей в корне отличается от восстановления транзакции и системы. Он включен в обсуждение для завершенности рассматриваемой темы.

Как уже отмечалось, *отказы носителей - это нарушения типа поломки головок дискового накопителя или дискового контроллера, когда некоторая часть базы данных разрушается физически*. Восстановление после такого нарушения включает перезапись (или *восстановление*) базы данных с резервной копии (или *дампа*) и последующее использование файла регистрации - как его активной, так и архивной части. В общем, такое восстановление - это повторное выполнение всех транзакций, вызванное применением резервной копии. При этом нет необходимости отмены транзакций, которые выполнялись в момент отказа носителей, поскольку по определению все обновления таких транзакций полностью утеряны.

Таким образом, восстановление средств подразумевает наличие *утилиты дамп/восстановление*. Дамп-часть этой утилиты используется для сохранения резервной копии. Такие копии могут сохраняться либо на ленте, либо другим архивным способом. После отказа носителей восстановительная часть утилиты используется для создания базы данных со специализированной резервной копии.

Двухфазная фиксация

Двухфазная фиксация важна всякий раз, когда определенная транзакция может взаимодействовать с несколькими независимыми *администраторами ресурсов*, каждый из которых руководит своим собственным набором восстанавливаемых ресурсов и поддерживает собственный файл регистрации (журнал). Например, пусть транзакция, запущенная в среде MVS компьютера IBM, модифицирует как базу данных IMS, так и базу данных DB2 (между прочим, такая транзакция вполне допустима). Если транзакция завершается успешно, то все ее обновления, как для данных IMS, так и для данных DB2, могут быть выполнены. В противном случае *все* ее обновления могут быть отменены или транзакция перестанет быть атомарной.

Для транзакции не имеет смысла выполнять COMMIT для IMS и ROLLBACK для DB2, и, даже если подобная инструкция будет выполнена для обеих баз, система все равно может дать сбой между двумя этими операциями. Вместо этого транзакция выполняет общесистемную команду COMMIT (или ROLLBACK). Этими операциями руководит системный компонент, называемый ***координатором***. Он гарантирует, что оба администратора ресурсов (т.е. IMS и DB2 в примере) передают или отменяют обновления, за которые они ответственны. Более того, он обеспечивает такую гарантию, *даже если система отказала в середине процесса*. Это происходит благодаря *протоколу двухфазной фиксации*.

Ниже приведена последовательность работы координатора. Для простоты примем, что транзакция в базе данных выполнена успешно, а значит, выдана системная операция COMMIT, а не ROLLBACK. После получения запроса на выполнение COMMIT координатор осуществляет следующий двухфазный процесс.

1. Во-первых, он дает указание всем администраторам ресурсов быть готовыми действовать на транзакцию "любым способом". На практике это означает, что каждый *участник* процесса, т.е. каждый администратор ресурсов, должен насильно сохранить все записи журнала (файла регистрации) для локальных ресурсов, используемых транзакцией вне собственных физических файлов регистрации (т.е. вне энергозависимой памяти). Теперь, что бы ни случилось, администратор ресурсов не будет выполнять *постоянной записи* от имени транзакции, а сможет при необходимости передавать свои обновления и отменять их. Если насильственная

запись прошла успешно, администратор ресурсов отвечает координатору, что все "OK". или, наоборот, - "Not OK".

2. Когда координатор получил соответствующие ответы от всех участников, он насильственно заносит записи в собственный физический файл регистрации, указывая свое решение относительно транзакции. Если все ответы были "OK", то решение будет "выполнить", а если был ответ "Not OK", то – "прокрутить назад". Затем координатор любым способом информирует каждого участника о своем решении, и *каждый участник согласно инструкции должен локально зафиксировать или аннулировать транзакции*. Отметьте, что каждый участник должен делать то, что ему велел координатор во время фазы 2, - в этом и состоит протокол. Обратите также внимание, что именно появление записи решения в физическом файле регистрации координатора и отмечает переход с фазы 1 на фазу 2.

Теперь, если система дает сбой в какой-либо точке во время полного процесса, процедура перезагрузки будет искать запись решения в файле регистрации координатора. Если она ее обнаружит, то сможет указать, где произошла остановка. Если эта запись не будет обнаружена, значит, принятым решением будет *откат* и, следовательно, процесс будет завершен.

Стоит подчеркнуть, что если координатор и участники выполняют свою работу на различных механизмах (поскольку они могут представлять собой распределенную систему), то ошибка в работе координатора может привести к тому, что некий участник довольно долго будет ожидать решения координатора. Во *время ожидания* ни одно обновление, произведенное транзакцией, не сможет произойти с помощью этого участника, оно будет как бы скрыто от других транзакций (иными словами, такое обновление будет *заблокировано*, об этом еще будет идти речь в следующих главах).

Отметим, что диспетчер передачи данных, также называемый администратором передачи данных, может считаться администратором ресурсов в описанном выше смысле. Это означает, что сообщения можно считать такими же восстанавливаемыми ресурсами, как и базу данных, а администратор передачи данных способен участвовать в процессе двухфазной фиксации.

Лекция 5. Функция параллелизм

Восстановление данных и параллельное выполнение операций следует рассматривать совместно, поскольку обе эти темы являются частями более общей темы, связанной с *обработкой транзакций*. Однако в этой главе основное внимание уделяется именно параллелизму. Термин ***параллелизм*** означает ***возможность одновременной обработки в СУБД многих транзакций с доступом к одним и тем же данным, причем в одно и то же время***. Как известно, в такой системе для корректной обработки параллельных транзакций без возникновения конфликтных ситуаций необходимо использовать некоторый ***метод управления параллелизмом***. Ниже приведены примеры конфликтных ситуаций, возможных при отсутствии соответствующего управления.

Вначале рассмотрим некоторые проблемы, которые могут возникнуть при отсутствии должного управления параллелизмом, а затем стандартный метод разрешения таких проблем, т.е. ***метод блокировки***.

Блокировка не является единственным возможным подходом в решении проблемы управления параллелизмом, но она, несомненно, чаще других встречается на практике. Однако с внедрением метода блокировки возникают другие проблемы; среди них наиболее известна проблема ***тупиковых ситуаций***, которая описывается ниже в этой главе. Затем описывается концепция ***способности к упорядочению***, которая определяется как формальный критерий правильности выполнения некоторого набора параллельных транзакций. Потом продолжается рассмотрение некоторых важных уточнений основной идеи блокирования, а именно ***уровней изоляции*** и ***преднамеренной блокировки***. Далее описывается, как используются перечисленные выше понятия в стандарте языка SQL.

Отметим, что идеи параллелизма, как и идеи восстановления данных, в некоторой степени не зависят от того, является ли СУБД реляционной или какой-либо другой. Однако большая часть теоретической работы в этой области, как и в области восстановления данных, выполнялась именно в некотором специализированном реляционном контексте.

Три проблемы параллелизма

Прежде всего следует уточнить, что каждый метод управления параллелизмом предназначен для решения некоторой конкретной задачи. Тем не менее, при обработке правильно составленных транзакций возникают ситуации, которые могут привести к получению неправильного результата из-за взаимных помех среди некоторых транзакций. При этом, вносящая помеху транзакция сама по себе может быть правильной. Неправильный конечный результат возникает по причине бесконтрольного *чередования* операций из двух правильных транзакций. Основные проблемы, возникающие при параллельной обработке транзакций следующие:

1. проблема *потери результатов обновления*;
2. проблема *незафиксированной зависимости*;
3. проблема *несовместимого анализа*.

Проблема потери результатов обновления

Рассмотрим ситуацию, показанную на рис. 5.1, в такой интерпретации: транзакция *A* извлекает некоторый кортеж *p* в момент времени t_1, транзакция *B* извлекает некоторый кортеж *p* в момент времени t_2, транзакция *A* обновляет некоторый кортеж *p* (на основе значений, полученных в момент времени t_1) в момент времени t_3, транзакция *B* обновляет тот же кортеж *p* (на основе значений, полученных в момент времени t_2, которые имеют те же значения, что и в момент времени t_1) в момент времени t_4. Однако результат операции обновления, выполненной транзакцией, будет утерян, поскольку в момент времени t_4 она не будет учтена и потому будет “отменена” операцией обновления, выполненной

транзакцией *B*.

Транзакция A	Время	Транзакция B
-	|	-
Извлечение кортежа *p*	t_1	-
-	|	-
-	t_2	Извлечение кортежа *p*
-	|	-
Обновление кортежа *p*	t_3	-
-	|	-
-	t_4	Обновление кортежа *p*
-	↓	-

Рис. 5.1 Потеря в момент времени t_4 результатов обновления,
выполненного транзакцией *A*

Проблема незафиксированной зависимости

Проблема незафиксированной зависимости появляется, если с помощью некоторой транзакции осуществляется извлечение (или, что еще хуже, обновление) некоторого кортежа, который в данный момент обновляется другой транзакцией, но это обновление еще не закончено. Таким образом, если обновление не завершено, существует некоторая вероятность того, что оно не будет завершено никогда. (Более того, в подобном случае может быть выполнен возврат к предыдущему состоянию кортежа с отменой выполнения транзакции.) В таком случае в первой транзакции будут принимать участие данные, которых больше не существует (в том смысле, что они "никогда" не существовали). Эта ситуация показана на рис. 5.2 и 5.3.

В первом примере (рис. 5.2) транзакция *A* в момент времени t_2 встречается с невыполненным обновлением (оно также называется невыполненным изменением). Затем это обновление отменяется в момент времени t_3. Таким образом, транзакция *A* выполняется на основе фальшивого предположения, что кортеж *p* имеет некоторое значение в момент времени t_2, тогда как на самом деле он имеет некоторое значение, существовавшее еще в момент времени t_1. В итоге после выполнения транзакции *A* будет получен неверный результат. Кроме того, обратите внимание, что отмена выполнения транзакции *B* может произойти не по вине транзакции *B*, а, например, в результате краха системы. (К этому времени выполнение транзакции *A* может быть уже завершено, а потому крушение системы не приведет к отмене выполнения транзакции.)

Транзакция A	Время	Транзакция B
-	|	-
-	t_1	Обновление кортежа *p*
-	|	-
Извлечение кортежа *p*	t_2	-
-	|	-
-	t_3	Отмена выполнения
-	↓	транзакции
		-

Рис. 5.2. Транзакция *A* становится зависимой от невыполненного
изменения в момент времени *t2*

Транзакция A	Время	Транзакция B
-	|	-
-	t_1	Обновление кортежа *p*
-	|	-
Обновление кортежа *p*	t_2	-
-	|	-
-	t_3	Отмена выполнения транзакции
-	↓	-

Рис. 5.3. Транзакция *A* обновляет невыполненное изменение в момент времени *t2*, и результаты этого обновления утрачиваются в момент времени *t3*

Второй пример, приведенный на рис. 5.3, иллюстрирует более худший случай. Не только транзакция *A* становится зависимой от изменения, не выполненного в момент времени t_2, но также в момент времени t_3 фактически утрачивается результат обновления, поскольку отмена выполнения транзакции *B* в момент времени t_3 приводит к восстановлению кортежа *p* к исходному значению в момент времени t_1. Это еще один вариант проблемы потери результатов обновления.

Проблема несовместимого анализа

На рис. 5.4 показаны транзакции *A* и *B*, которые выполняются для кортежей со счетами (СЧЕТ). При этом транзакция *A* суммирует балансы, транзакция *B* производит перевод суммы 10 со счета 3 на счет 1. Полученный в итоге транзакции *A* результат 110, очевидно, неверен, и если он будет записан в базе данных, то в ней может возникнуть проблема несовместимости. В таком случае говорят, что транзакция *A* встретилась с несовместимым состоянием и на его основе был выполнен несовместимый анализ. Обратите внимание на следующее различие между этим примером и предыдущим: здесь не идет речь о зависимости транзакции *A* от транзакции *B*, так как транзакция *B* выполнила все обновления до того, как транзакция *A* извлекла СЧЕТ 3.

Блокировка

Описанные выше проблемы могут быть разрешены с помощью методики управления параллельным выполнением процессов под названием ***блокировка*** или ***синхронизационные захваты***. Ее основная идея очень проста: в случае, когда для выполнения некоторой транзакции необходимо, чтобы некоторый объект (обычно это кортеж базы данных) не изменялся непредсказуемо и без ведома этой транзакции (как это обычно бывает), такой объект ***блокируется***. Таким образом, эффект блокировки состоит в том, чтобы “заблокировать доступ к этому объекту со стороны других транзакций”, а значит, предотвратить непредсказуемое изменение этого объекта. Следовательно, первая транзакция в состоянии выполнить всю необходимую обработку с учетом того, что обрабатываемый объект остается в стабильном состоянии настолько долго, насколько это нужно.

Описание функционирования блокировки:

1. Прежде всего, предположим, что в системе поддерживается два типа блокировок: ***блокировка без взаимного доступа (монопольная блокировка)***, называемая X-блокировкой (X locks - eXlusive locks), и ***блокировка с взаимным доступом***, называемая S-блокировкой (S locks - Shared locks). Надо заметить, что X- и S-блокировки иногда называют блокировками записи и чтения соответственно. Здесь предполагается, что X- и S-блокировки единственно возможные, хотя ниже в этой главе приводятся примеры блокировок других типов. Кроме того, допустим, что в данном случае кортежи являются единственным типом

"блокируемого объекта", хотя опять же ниже описаны примеры других типов таких объектов.

2. Если транзакция *A* блокирует кортеж *p* без возможности взаимного доступа (X-блокировка), то запрос другой транзакции *B* с блокировкой этого кортежа *p* будет отменен.
3. Если транзакция *A* блокирует кортеж *p* с возможностью взаимного доступа (S-блокировка), то

- запрос со стороны некоторой транзакции *B* на X-блокировку кортежа будет отвергнут;
- запрос со стороны некоторой транзакции *B* на S-блокировку кортежа *p* будет принят (т.е. транзакция *B* также будет блокировать кортеж *p* с помощью S-блокировки.

Счет 1 40	Счет 2 50	Счет 3 30
Транзакция A	Время	Транзакция B
-	|	-
-	|	-
Извлечение кортежа	t_1	-
СЧЕТ1:	|	-
sum = 40	|	-
-	t_2	-
Извлечение кортежа	|	-
СЧЕТ2:	t_3	Извлечение кортежа
sum = 90	|	СЧЕТ 3:
-	t_4	-
-	|	Обновление кортежа
-	|	СЧЕТ 3:
-	t_5	30 → 20
-	|	-
-	t_6	Извлечение кортежа
-	|	СЧЕТ 1:
-	|	-
-	t_7	Обновление кортежа
-	|	СЧЕТ 1:
-	|	40 → 50
-	t_8	-
-	↓	Завершение
Извлечение кортежа		выполнения транзакций
СЧЕТ 3:		
sum = 110 (а не 120)		

Рис. 5.4 Транзакция *A* выполнила несовместимый анализ

Эти правила можно наглядно представить в виде матрицы совместимости, показанной на рис. 5.5, и интерпретировать ее следующим образом. Рассмотрим некоторый кортеж *p* и предположим, что транзакция *A* блокирует кортеж *p* различными типами блокировки (это обозначено соответствующими символами S и X, а отсутствие блокировки - прочерком). Предположим также, что некоторая транзакция *B* запрашивает блокировку кортежа *p*, что обозначено в первом слева столбце матрицы на рис. 5.5 (для полноты картины в таблице также приведен случай "отсутствия блокировки"). В других ячейках матрицы символ N обозначает конфликтную ситуацию (запрос со стороны транзакции *B* не может быть удовлетворен, и сама эта транзакция переходит в состояние ожидания), а *Y* - полную совместимость (запрос со стороны транзакции *B* удовлетворен).

Очевидно, что эта матрица является симметричной.

	X	S	-
X	N	N	Y
S	N	Y	Y

Рис. 5.5. Матрица совместимости для X- и S-блокировки

Теперь следует ввести ***протокол доступа к данным***, который на основе введения только что описанных X- и S-блокировки позволяет избежать возникновения проблем параллелизма (описанных выше в этой главе).

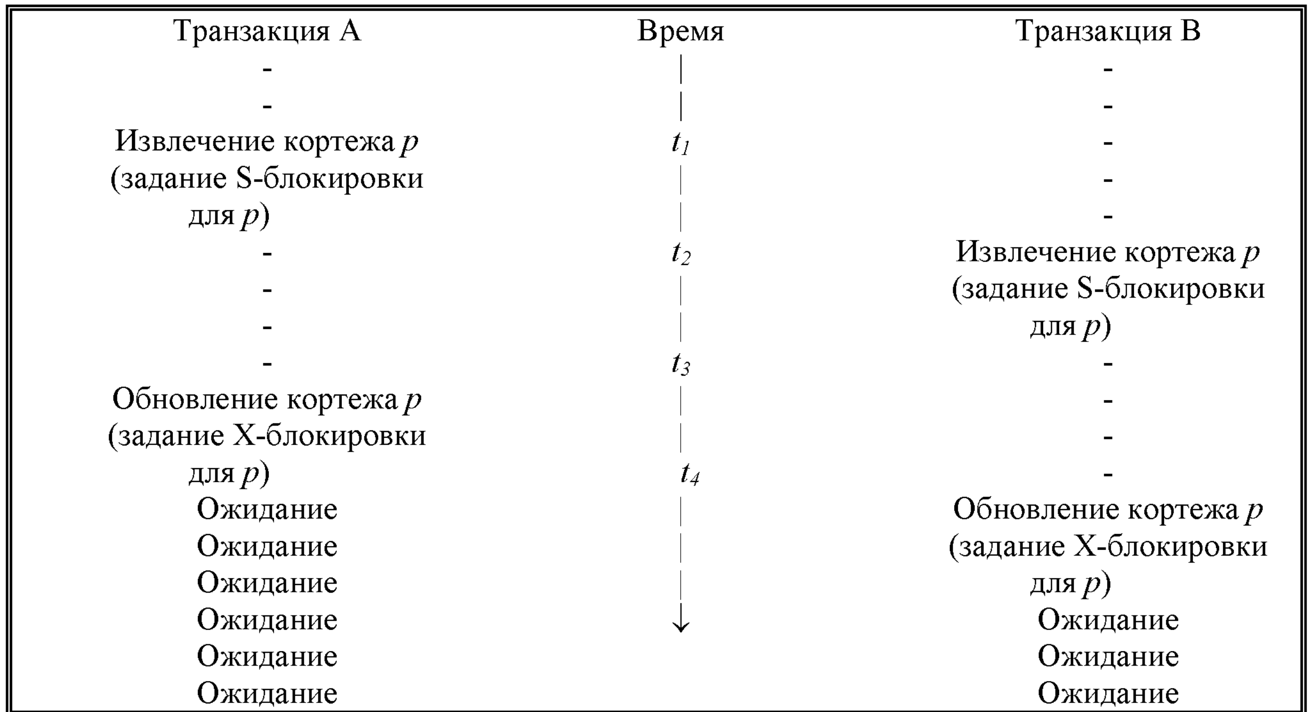
1. Транзакция, предназначенная для извлечения кортежа, прежде всего должна наложить S-блокировку на этот кортеж.
2. Транзакция, предназначенная для обновления кортежа, прежде всего должна наложить X-блокировку на этот кортеж. Иначе говоря, если, например, для последовательности действий типа извлечение/обновление для кортежа уже задана S-блокировка, то ее необходимо *заменить* X-блокировкой.
3. Если запрашиваемая блокировка со стороны транзакции *B* отвергается из-за конфликта с некоторой другой блокировкой со стороны транзакции *A*, то транзакция *B* переходит в состояние *ожидания*. Причем транзакция *B* будет находиться в состоянии ожидания до тех пор, пока не будет снята блокировка, заданная транзакцией *A*. Надо отметить, что в системе обязательно должны быть предусмотрены способы устранения бесконечно долгого состояния ожидания транзакции *B* (это состояние обычно называется ***зависанием***). Самым простым способом может служить организация обработки запросов на блокировку в очередности "первым поступил - первым обработан".
4. X-блокировки сохраняются вплоть до конца выполнения транзакции (до операции "завершение выполнения" или "отмена выполнения"). S-блокировки также обычно сохраняются вплоть до этого момента, однако при работе с ними следует учесть замечания, которые излагаются несколько ниже в этой лекции.

Замечание. Блокировки в транзакциях обычно задаются неявным образом: например, запрос на "извлечение кортежа" является неявным запросом с S-блокировкой, а запрос на "обновление кортежа" - неявным запросом с X-блокировкой соответствующего кортежа. При этом под термином "обновление" (как и ранее) подразумеваются помимо самих операций обновления также операции вставки и удаления. При строгом описании протокола возможны небольшие отличия, связанные с операциями вставки и удаления, однако здесь они будут опущены.

Решение проблем параллелизма

Проблема потери результатов обновления

На рис. 5.6 приведена измененная версия процесса, показанного на рис. 5.1, с учетом применения протокола блокировки для чередующихся операций. Операция обновления для транзакции *A* в момент времени t_3 не будет выполнена, поскольку она является неявным запросом с заданием X-блокировки для кортежа *p*, а этот запрос вступает в конфликт с S-блокировкой, уже заданной транзакцией *B*. Таким образом, транзакция *A* переходит в состояние ожидания. По аналогичным причинам транзакция *B* переходит в состояние ожидания в момент времени t_4. Хотя в таком случае результаты любых обновлений не будут утрачены, решение этой проблемы с помощью блокировки возможно только при решении другой проблемы. Эта новая проблема называется ***тупиком***.

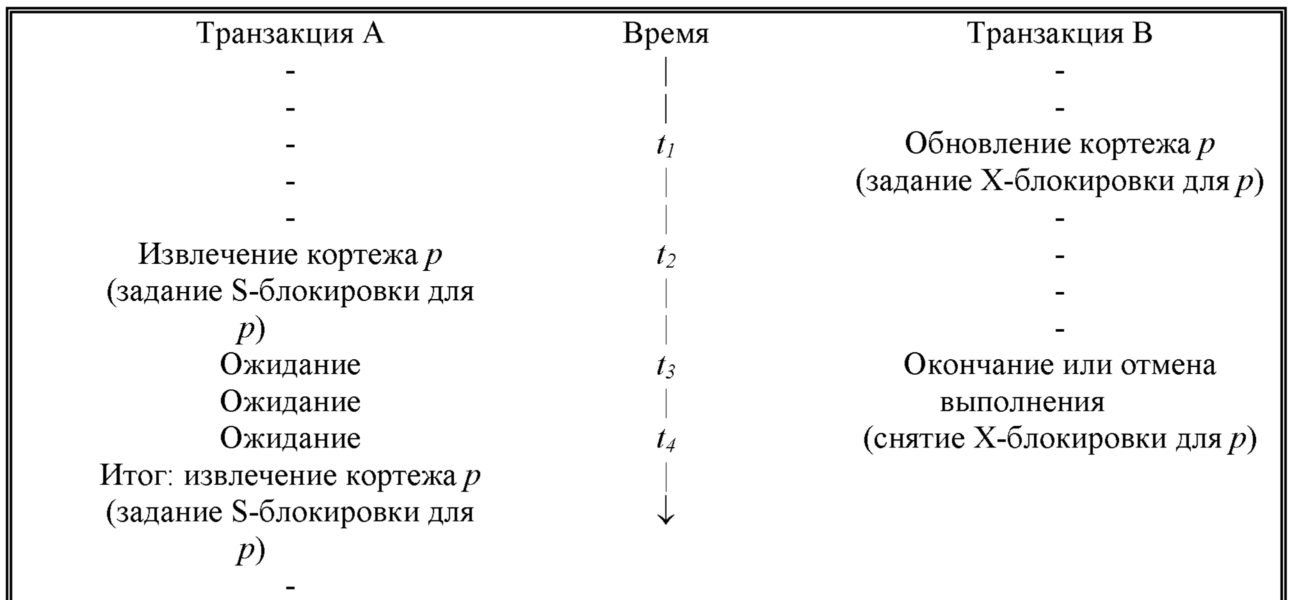


Транзакция A	Время	Транзакция B
-		-
-		-
Извлечение кортежа *p* (задание S-блокировки для *p*)	t_1	-
-		-
-	t_2	Извлечение кортежа *p* (задание S-блокировки для *p*)
-		-
-	t_3	-
Обновление кортежа *p* (задание X-блокировки для *p*)	t_4	-
Ожидание		Обновление кортежа *p* (задание X-блокировки для *p*)
Ожидание		
Ожидание		
Ожидание	↓	Ожидание
Ожидание		Ожидание
Ожидание		Ожидание

Рис. 5.6. Хотя обновления не утрачиваются, но в момент времени t_4 возникает тупиковая ситуация

Проблема незафиксированной зависимости

На рис. 5.7 и 5.8 приведены в измененном виде примеры, показанные ранее на рис. 5.2 и 5.3 соответственно. Они демонстрируют чередующееся выполнение операций согласно описанному выше протоколу блокировки. Операция для транзакции *A* в момент времени t_2 (извлечение на рис. 5.7 и обновление на рис. 5.8) не будет выполнена. Дело в том, что она является неявным запросом с заданием блокировки для кортежа *p*, а этот запрос вступает в конфликт с X-блокировкой, уже заданной транзакцией *B*.

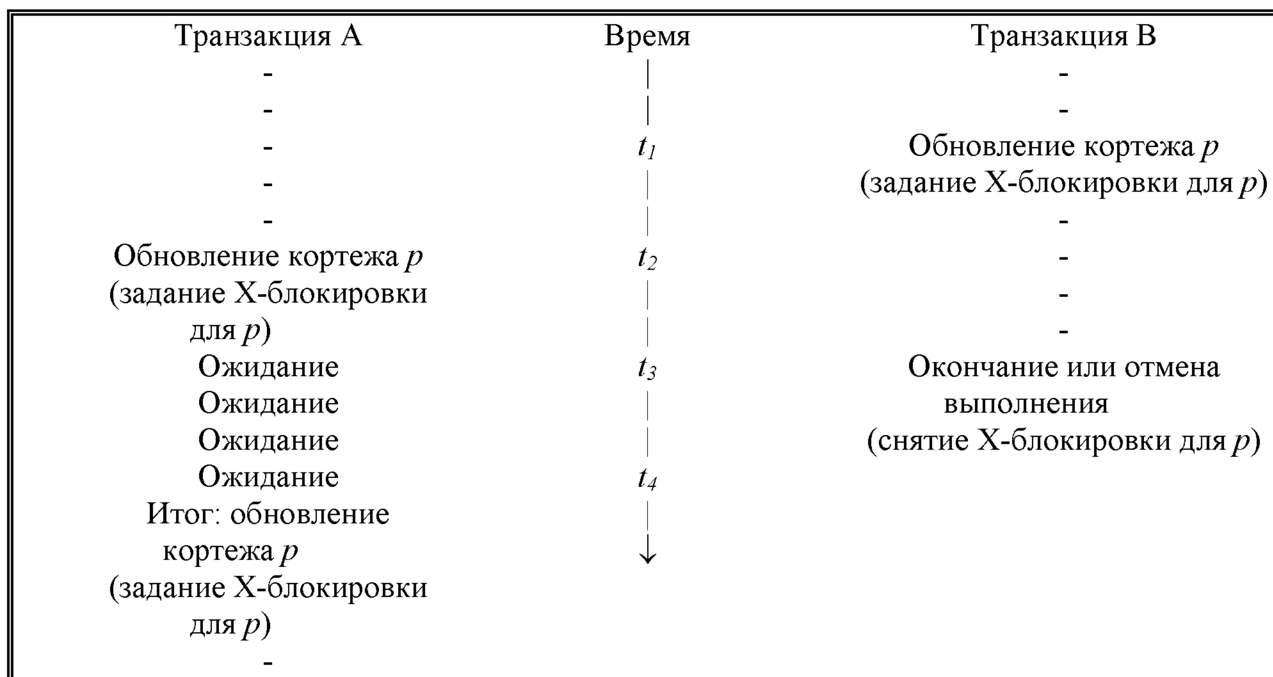


Транзакция A	Время	Транзакция B
-		-
-		-
-	t_1	Обновление кортежа *p* (задание X-блокировки для *p*)
-		-
Извлечение кортежа *p* (задание S-блокировки для *p*)	t_2	-
		-
		-
Ожидание	t_3	Окончание или отмена выполнения
Ожидание		
Ожидание	t_4	(снятие X-блокировки для *p*)
Итог: извлечение кортежа *p* (задание S-блокировки для *p*)	↓	
-		

Рис. 5.7. Транзакция *A* предохраняется от выполнения операций с незафиксированным изменением в момент времени t_2

Таким образом, транзакция *A* переходит в состояние ожидания до тех пор, пока не будет прекращено выполнение транзакции *B* (до операции окончания или отмены вы-

полнения транзакции *B*). Тогда заданная транзакцией *B* блокировка будет снята и транзакция *A* может быть выполнена. Причем транзакция *A* будет иметь дело с некоторым *фиксированным* значением (либо существовавшим до выполнения транзакции *B* при отмене ее выполнения, либо полученным после выполнения транзакции *B*). В любом случае транзакция *A* больше не зависит от незафиксированного обновления.



Транзакция A	Время	Транзакция B
-		-
-		-
-	t_1	Обновление кортежа *p*
-		(задание X-блокировки для *p*)
-		-
Обновление кортежа *p*	t_2	-
(задание X-блокировки		-
для *p*)		-
Ожидание	t_3	Окончание или отмена
Ожидание		выполнения
Ожидание		(снятие X-блокировки для *p*)
Ожидание	t_4	
Итог: обновление		
кортежа *p*	↓	
(задание X-блокировки		
для *p*)		
-		

Рис. 5.8. Транзакция *A* предохраняется от выполнения операций с незафиксированным изменением в момент времени t_2

Проблема несовместимого анализа

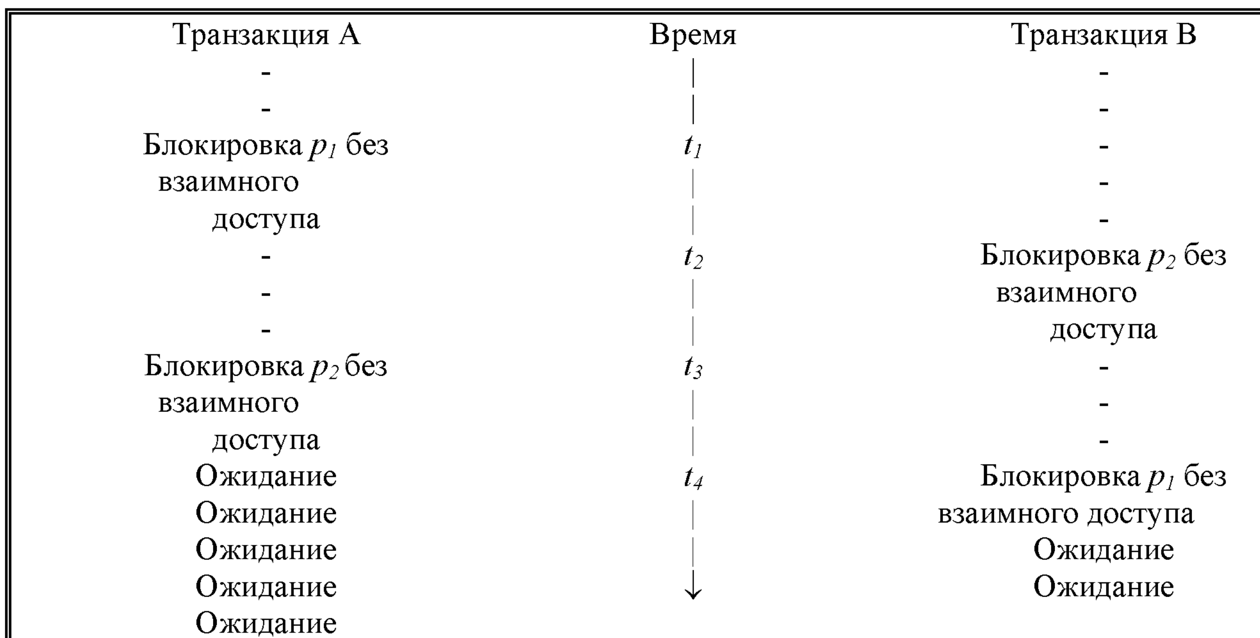
На рис. 5.9 приведена измененная версия рис. 5.4 с перечислением чередующихся транзакций согласно протоколу блокировки из раздела 5.3. Операция обновления для транзакции *B* в момент времени t_6 не будет выполнена. Дело в том, что она является неявным запросом с заданием X-блокировки для кортежа СЧЕТ 1, а этот запрос вступает в конфликт с S-блокировкой, уже заданной транзакцией *A*. Таким образом, транзакция *B* переходит в состояние ожидания. Точно так же операция извлечения для транзакции *A* в момент времени t_7 не будет выполнена. Дело в том, что она является неявным запросом с заданием S-блокировки для кортежа СЧЕТ 3, а этот запрос вступает в конфликт с X-блокировкой, уже заданной транзакцией *B*. Таким образом, транзакция *A* переходит в состояние ожидания. Следовательно, блокировка хотя и помогает решить одну проблему (а именно проблему несовместимого анализа), но приводит к необходимости решения другой проблемы (а именно проблемы возникновения тупиковой ситуации, которая обсуждается в ниже).

Тупики, их обнаружение и распознавание

Как было показано выше, блокировку можно использовать для разрешения трех основных проблем, возникающих при параллельной обработке кортежей. К сожалению, использование блокировок приводит к возникновению другой проблемы - тупиковой ситуации. Два примера таких ситуаций были приведены выше. На рис. 5.10 показан обобщенный пример этой проблемы, в котором p_1 и p_2 представляют любые блокируемые объекты, необязательно кортежи базы данных, а выражения типа “блокировка ... без взаимного доступа” представляют любые операции с наложением блокировки (без взаимного доступа), заданные как явно, так и неявно.

СЧЕТ 1 40	СЧЕТ 2 50	СЧЕТ 3 30
Транзакция А	Время	Транзакция В
-	│	-
-	│	-
Извлечение кортежа	t_1	-
СЧЕТ 1:	│	-
(задание S-блокировки	│	-
для	│	-
кортежа СЧЕТ 1)	│	-
sum = 40	t_2	-
-	│	-
Извлечение кортежа	│	-
СЧЕТ 2	│	-
(задание S-блокировки	│	-
для	t_3	Извлечение кортежа СЧЕТ 3:
кортежа СЧЕТ 2)	│	(задание S-блокировки для
sum = 90	│	кортежа СЧЕТ 3)
-	│	-
-	t_4	Обновление кортежа СЧЕТ 3:
-	│	(задание X-блокировки для
-	│	кортежа СЧЕТ 3)
-	│	30→20
-	│	-
-	t_5	Извлечение кортежа СЧЕТ 1:
-	│	(задание S-блокировки для
-	│	кортежа СЧЕТ1)
-	│	-
-	t_6	Обновление кортежа СЧЕТ1:
-	│	(задание X-блокировки для
-	│	кортежа СЧЕТ 1)
-	│	Ожидание
Извлечение кортежа	t_7	Ожидание
СЧЕТ 3:	│	Ожидание
(задание S-блокировки	│	Ожидание
для	│	Ожидание
кортежа СЧЕТ 3)	↓	Ожидание
Ожидание		
Ожидание		

Рис. 5.9. Проблема несовместимого анализа разрешается,
но в момент времени t_7 возникает тупиковая ситуация



Транзакция A	Время	Транзакция B
-	|	-
-	|	-
Блокировка p_1 без	t_1	-
взаимного	|	-
доступа	|	-
-	t_2	Блокировка p_2 без
-	|	взаимного
-	|	доступа
Блокировка p_2 без	t_3	-
взаимного	|	-
доступа	|	-
Ожидание	t_4	Блокировка p_1 без
Ожидание	|	взаимного доступа
Ожидание	|	Ожидание
Ожидание	↓	Ожидание
Ожидание		

Рис. 5.10. Пример тупиковой ситуации

Тупиковая ситуация возникает тогда, когда две или более транзакции одновременно находятся в состоянии ожидания, причем для продолжения работы каждая из транзакций ожидает прекращения выполнения другой транзакции. На рис. 5.10 показана тупиковая ситуация, включающая две транзакции, однако в принципе возможны тупиковые ситуации с участием трех, четырех и более транзакций. Однако эксперименты с системой System R показали, что на практике никогда не встречаются тупиковые ситуации с участием более чем двух транзакций.

Желательно, чтобы при возникновении тупиковой ситуации система могла обнаружить ее и найти из нее выход. Для обнаружения тупиковой ситуации следует обнаружить цикл в ***диаграмме состояний ожидания***, т.е. в перечне "транзакций, которые ожидают окончания выполнения других транзакций". Поиск выхода из тупиковой ситуации состоит в выборе одной из заблокированных транзакций в качестве жертвы и отмене ее выполнения. Таким образом, с нее снимается блокировка, а выполнение другой транзакции может быть возобновлено.

На практике не все системы в состоянии обнаружить тупиковую ситуацию. Например, в некоторых из них используется хронометраж выполнения транзакций, и сообщение о возникновении тупиковой ситуации поступает, если транзакция не выполняется за некоторое предписанное заранее время.

Рассмотрим различные методы разрешения тупиковых ситуаций.

Способность к упорядочению

В предыдущих разделах были описаны некоторые основы, необходимые для объяснения ключевого понятия ***"способность к упорядочению"***, которое является общепринятым ***критерием правильности*** управления параллельной обработкой кортежей. Точнее говоря, чередующееся выполнение заданного множества транзакций будет верным, если оно упорядочено, т.е. при его выполнении будет получен такой же результат, как и при *последовательном* выполнении тех же транзакций. Обосновать это утверждение помогут следующие замечания:

1. Отдельные транзакции считаются верными, если при их выполнении база данных переходит из одного непротиворечивого состояния в другое непротиворечивое со-

стояние в смысле.

2. Следовательно, выполнение транзакций одна за другой в любом последовательном порядке также является верным. При этом под выражением "любой последовательный порядок" подразумевается, что используются независимые друг от друга транзакции.
3. Чередующееся выполнение транзакций, следовательно, является верным, если оно эквивалентно некоторому последовательному выполнению, т.е. если оно подлежит упорядочению.

Понятно, что для того, чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций. В последнее время вместо термина «**упорядочение**» транзакций зачастую используют термин «**сериализация**» транзакций. План (способ) выполнения набора транзакций называется **сериальным**, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.

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

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

Возвращаясь к приведенным выше примерам (рис. 5.1 - 5.4), можно отметить: данная проблема в каждом случае заключалась в том, что чередующееся выполнение транзакций не было упорядочено, т.е. *не* было эквивалентно выполнению либо сначала транзакции *A*, а затем транзакции *B*, либо сначала транзакции *B*, а затем транзакции *A*. Основное значение предложенной выше схемы блокировки заключается в *принудительном* упорядочении. На рис. 5.7 и 5.8 чередующееся выполнение эквивалентно выполнению сначала транзакции *B*, а затем транзакции *A*. На рис. 5.6 и 5.9 показана тупиковая ситуация, в которой можно было бы отменить и, возможно, повторно запустить одну из двух транзакций. Если транзакция *A* отменяется, чередующееся выполнение вновь становится эквивалентным выполнению сначала транзакции *B*, а затем транзакции *A*.

В данном курсе будем использовать следующую терминологию. Для заданного набора транзакций любой порядок их выполнения (чередующийся или какой-либо другой) называется ***графиком запуска*** вместо плана. Выполнение транзакций по одной без их чередования называется ***последовательным графиком запуска***, а непоследовательное выполнение транзакций - ***чередующимся графиком запуска*** или непоследовательным графиком запуска. Два графика называются ***эквивалентными***, если при их выполнении будет получен *одинаковый* результат, независимо от исходного состояния базы данных. Таким образом, график запуска является верным (т.е. допускающим возможность упорядочения), если он эквивалентен некоторому последовательному графику запуска.

Стоит подчеркнуть, что при выполнении двух различных последовательных графиков запуска, содержащих одинаковый набор транзакций, можно получить совершенно различные результаты. Поэтому выполнение двух различных чередующихся графиков запуска с одинаковыми транзакциями может также привести к различным результатам, которые могут быть восприняты как верные. Например, предположим, что транзакция *A* означает действие "сложить 1 с *x*", а транзакция *B*- "удвоить *x*" (где *x* - это некоторый объект базы данных). Предположим также, что начальное значение *x* равно 10. Тогда при последовательном выполнении сначала транзакции *A*, а затем транзакции *B* будет получен

результат x=22, а при последовательном выполнении сначала транзакции *B*, а затем транзакции *A* - x=21. Оба результата одинаково верные, а любой график запуска, эквивалентный выполнению либо сначала транзакции *A*, а затем транзакции *B*, либо сначала транзакции *B*, а затем транзакции *A*, также является верным.

Концепция способности к упорядочению была впервые предложена (хотя и под другим названием) Есвараном (Eswaran). В его работе также доказана важная теорема ***двухфазной блокировки***, которая кратко может быть сформулирована следующим образом:

Если все транзакции подчиняются "протоколу двухфазной блокировки", то для всех возможных чередующихся графиков запуска существует возможность упорядочения.

При этом ***протокол двухфазной блокировки***, в свою очередь, формулируется следующим образом.

1. Перед выполнением каких-либо операций с некоторым объектом (например, с кортежем базы данных) транзакция должна заблокировать этот кортеж.
2. После снятия блокировки транзакция не должна накладывать никаких других блокировок.

Таким образом, транзакция, которая подчиняется этому протоколу, характеризуется двумя фазами: фазой наложения блокировки и фазой снятия блокировки.

На практике вторая фаза часто сводится к единственной операции окончания выполнения (или отмены выполнения) в конце транзакции. Действительно, описанный выше в этой главе протокол блокировки можно рассматривать как строгую формулировку двухфазного протокола.

Понятие способности к упорядочению существенно облегчает понимание этой довольно запутанной области знаний. В дополнение к сказанному следует добавить несколько комментариев. Пусть *E* является чередующимся графиком запуска, включающим некоторый набор транзакций T_1, T_2,..., T_n. Если *E* является графиком, допускающим возможность упорядочения, то существует некоторый последовательный график запуска *S*, содержащий такой набор транзакций T_1, T_2, ... T_n, что график *E* эквивалентен графику *S*. В таком случае *S* называется ***упорядочением*** *E*. Как уже было показано ранее, *S* необязательно является уникальным, т.е. некоторый график запуска *E* может иметь несколько упорядочений.

Пусть T_i и T_j - некоторые транзакции множества транзакций T_1, T_2, ..., T_n. Не теряя общности изложения, предположим, что в упорядочении *S* транзакция T_i предшествует T_j. Следовательно, для чередующегося графика запуска *E* это значит, что транзакция T_i выполняется перед транзакцией T_j. Иначе говоря, неформальная, но очень полезная характеристика упорядочения может быть выражена следующим образом. Если *A* и *B* являются любыми двумя транзакциями некоторого графика запуска, допускающего возможность упорядочения, то либо *A* логически предшествует *B*, либо *B* логически предшествует *A*, т.е. ***либо B использует результаты выполнения транзакции A, либо A использует результаты выполнения транзакции B***. (Если транзакция *A* приводит к обновлению кортежей *p, q, ... r* и транзакция *B* использует эти кортежи в качестве входных данных, то используются либо все обновленные с помощью *A* кортежи, либо полностью не обновленные кортежи до выполнения транзакции *A*, но никак не их смесь.) Наоборот, график запуска является неверным и не подлежит упорядочению, если результат выполнения транзакций не соответствует либо сначала выполнению транзакции *A*, а затем транзакции *B*, либо сначала выполнению транзакции *B*, а затем транзакции *A*.

В заключение стоит подчеркнуть, что если некоторая транзакция *A* не является двухфазной (т.е. не удовлетворяет протоколу двухфазной блокировки), то всегда можно построить некоторую другую транзакцию *B*, которая при чередующемся выполнении вместе с транзакцией *A* может привести к графику запуска, не подлежащему упорядочению и неверному. В настоящее время с целью понижения требований к ресурсам и, следовательно, повышения производительности и пропускной способности в

реальных системах обычно предусмотрено использование не двухфазных транзакций, а транзакций с "ранним снятием блокировки" (еще до выполнения операции прекращения транзакции) и наложением нескольких блокировок. Однако следует понимать, что использование таких транзакций сопряжено с большим риском. Действительно, при использовании недвухфазной транзакции *A* предполагается, что в данной системе не существует никакой другой чередующейся с ней транзакции *B* (в противном случае в системе возможно получение ошибочных результатов).

Уровни изоляции

Термин ***уровень изоляции***, грубо говоря, используется для описания *степени вмешательства* параллельных транзакций в работу некоторой заданной транзакции. Но при обеспечении возможности упорядочения не допускается никакого вмешательства, иначе говоря, уровень изоляции должен быть максимальным. Однако, как уже отмечалось в конце предыдущего раздела, в реальных системах по различным причинам обычно допускаются транзакции, которые работают на уровне изоляции ниже максимального.

Замечание. Уровень изоляции обычно рассматривается как некоторое свойство *транзакции*. В действительности нет никаких причин, по которым данная транзакция не могла бы работать в одно и то же время на различных уровнях изоляции в разных частях базы данных. Однако здесь для простоты уровень изоляции будет рассматриваться всего лишь как некоторое свойство транзакции.

Уровней изоляции может быть несколько, например, в стандарте языка SQL, - четыре, а в системе DB2 фирмы IBM поддерживается два уровня. Вообще говоря, чем выше уровень изоляции, тем меньше степень вмешательства (и параллелизма), а чем ниже уровень изоляции, тем больше степень вмешательства (и параллелизма). В качестве примера рассмотрим два уровня, поддерживаемых системой DB2, которые называются ***уровнями стабильности курсора*** *и* ***повторяемого считывания***. Уровень *повторяемого считывания* (ПС) является максимальным уровнем; причем если все транзакции действуют на этом уровне, то графики запуска обладают возможностью упорядочения (при изложении материала выше по умолчанию полагалось, что все транзакции выполняются именно на этом уровне изоляции). Уровню *стабильности курсора* (СК) для транзакции T_1 присущи несколько другие особенности:

- транзакция адресуется к некоторому кортежу *p*,
- таким образом задается блокировка для кортежа *p*,
- после этого снимается адресуемость к кортежу *p* без его обновления,
- уровень X-блокировки не достигается,
- в результате блокировка снимается без необходимости ожидания окончания выполнения транзакции.

Обратите внимание, что теперь некоторая другая транзакция T_2 может привести к обновлению кортежа *p* и внесению в него изменений. Если транзакция T_1 вновь обратится к кортежу *p*, эти изменения будут обнаружены, что может привести к несовместимому состоянию базы данных. На уровне ПС, наоборот, все блокировки кортежа (а не только X-блокировки) сохраняются до окончания выполнения транзакции и упомянутой выше проблемы не возникает.

Замечание. Описанная проблема *не* единственная, которая может возникнуть на уровне СК, просто ее легче всего объяснить. Однако она, к сожалению, приводит к выводу, что уровень ПС необходим только в сравнительно маловероятных случаях, когда данная транзакция дважды выполняется для одного и того же кортежа. Однако существуют аргументы в защиту *повсеместного* использования уровня ПС по сравнению с уровнем СК. Дело в том, что выполнение транзакции на уровне СК не является двухфазным, а потому (как разъяснялось выше) в этом случае возможность упорядочения не гарантируется. В качестве контраргумента следует указать тот факт, что на уровне СК

достигается более широкий параллелизм, чем на уровне ПС (такая ситуация возможна, но вовсе не обязательна).

В заключение следует отметить, что предшествующая характеристика уровня ПС как уровня максимальной изоляции относится к уровню ПС, воплощенному в системе DB2. К сожалению, в стандарте языка SQL тот же термин “повторяемость считывания” используется для описания уровня изоляции, который находится ниже максимального уровня.

Протокол преднамеренной блокировки или гранулированные синхронизационные захваты

До сих пор в этой лекции предполагалось, что блокировке подвергается отдельный кортеж. Однако в принципе не существует никаких ограничений на блокирование больших или меньших единиц данных, например целого отношения, базы данных или (пример противоположного характера) некоторого значения атрибута внутри заданного кортежа. Ситуация такого типа называется ***степенью дробления блокировок***. Как обычно, здесь наблюдается некоторый тонкий баланс между степенью дробления и параллелизмом: чем мельче степень дробления, тем выше параллелизм, чем крупнее степень дробления, тем меньше блокировок можно задать и требуется протестировать, а значит, будут меньшие накладные расходы (т.е. требуется меньшее количество необходимых действий). Например, если транзакция накладывает X-блокировку на целое отношение, то нет необходимости задавать X-блокировку для отдельных кортежей внутри этого отношения, что в результате приводит к уменьшению общего числа блокировок. С другой стороны, никакая другая параллельная транзакция не в состоянии наложить никаких других блокировок на это отношение или кортежи этого отношения.

Предположим, что транзакция *T* задает X-блокировку для некоторого отношения *R*. В ответ на запрос транзакции *T* система должна сообщить о наличии других блокировок, наложенных другими транзакциями на любой кортеж отношения *R*. И если такая блокировка наложена, то запрос транзакции *T* не будет выполнен в данный момент времени. Как система может обнаружить конфликт такого рода? Очевидно, было бы крайне нежелательно проверять, блокируется ли каждый кортеж отношения *R* какой-либо другой транзакцией, а также задана ли какая-нибудь блокировка для какого-либо кортежа отношения *R*. Вместо этого можно ввести еще один протокол, а именно ***протокол преднамеренной блокировки или гранулированных синхронизационных захватов***, в соответствии с которым, прежде чем накладывать блокировку на кортеж (вероятно, преднамеренную блокировку, о которой подробнее рассказывается ниже), следует наложить ее на все отношения, в которых этот кортеж находится. Тогда обнаружить конфликтную ситуацию, рассмотренную выше, значительно проще за счет обнаружения блокировки на *уровне отношений*.

Как уже упоминалось, X- и S-блокировки синхронизационного захвата можно задавать как для отдельных кортежей, так и для целых отношений. Можно ввести три типа преднамеренных блокировок (гранул), которые имеют смысл для целых отношений, но не для отдельных кортежей:

- ***преднамеренная блокировка с возможностью взаимного доступа (Intent Shared lock - IS);***
- ***преднамеренная блокировка без взаимного доступа (Intent eXclusive lock – IX);***
- ***преднамеренная блокировка одновременно с возможностью взаимного доступа;***
- ***преднамеренная блокировка без возможности взаимного досутпа (Shared Intent eXclusive lock - SIX)***.

Ниже приводятся неформальные определения различных видов блокировок (здесь предполагается, что транзакция *T* накладывает блокировку рассматриваемого типа на отношение *R*), причем для полноты картины также приводятся определения S- и X-

блокировки.

- IS - транзакция *T* накладывает S-блокировки на отдельные кортежи отношения *R* для того, чтобы гарантировать стабильность этих кортежей при их обработке.
- IX - в дополнение к действиям, описанным в приведенной выше формулировке, транзакция *T* может осуществить обновление отдельных кортежей отношения *R*, а потому на эти кортежи накладываются X-блокировки.
- S - транзакцией *T* допускаются параллельные считывания для отношения *R*, но не обновления. Сама по себе транзакция *T* не может обновлять любые кортежи отношения *R*.
- SIX - в определении этой блокировки комбинируются определения S- и IX-блокировки, т.е. транзакцией *T* допускаются параллельные считывания для отношения *R*, но не обновления. *В дополнение* к этому транзакция *T* может осуществить обновление отдельных кортежей отношения *R*, а потому на эти кортежи накладываются X-блокировки.
- X - Транзакцией *T* вовсе не допускаются параллельные запросы к отношению *R*. Сама по себе транзакция *T* либо может, либо не может обновлять любые кортежи отношения *R*.

Формальные определения этих типов блокировок можно дать с помощью показанной на рис. 5.11 матрицы совместимости, которая является расширенной версией матрицы, представленной выше.

	X	SIX	IX	S	IS	-
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y

Рис. 5.11. Матрица совместимости, расширенная преднамеренными блокировками

Теперь можно представить более точную формулировку протокола преднамеренной блокировки.

1. Прежде чем транзакция наложит S-блокировку на данный кортеж, она должна наложить IS-блокировку или другую более сильную блокировку на отношение, в котором содержится данный кортеж.
2. Прежде чем транзакция наложит X-блокировку на данный кортеж, она должна наложить IX-блокировку или другую более сильную блокировку на отношение, в котором содержится данный кортеж.

Упомянутое понятие *относительной силы блокировок* можно объяснить с помощью ***диаграммы приоритета***, представленной на рис. 5.12. Блокировка типа L_2 называется более сильной (т.е. находится выше на диаграмме приоритета) по отношению к блокировке L_1 тогда и только тогда, когда для любой конфликтной ситуации (N) в столбце блокировки L_1 в некоторой строке матрицы совместимости существует также конфликт в столбце блокировки L_2 в той же строке (см. рис. 5.11). Обратите внимание, что запрос на задание блокировки, который отвергается для некоторого типа блокировки, также будет отвергнут и для более сильного типа блокировки (таким образом подразумевается, что всегда можно использовать типы блокировки, более сильные по сравнению с той, которая строго необходима в некотором заданном случае).

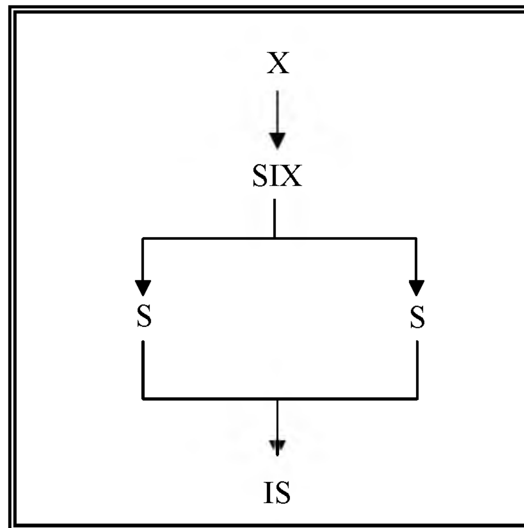


Рис. 5.12. Диаграмма приоритета различных типов блокировок

Стоит отметить, что (согласно протоколу преднамеренной блокировки) на уровне отношений блокировки обычно задаются неявным образом. Например, для транзакции считывания система может неявным образом наложить IS-блокировку на каждое отношение, к которому обращается данная транзакция. А для транзакции обновления вместо этого, вероятно, потребуется задать IX-блокировку. Однако в системе также может быть предусмотрено явное задание блокировок с помощью разного рода утверждений LOCK, для того чтобы в случае необходимости позволить транзакциям наложить S-, X- и SIX-блокировки на заданном уровне отношений. Например, такое заданное явным образом утверждение LOCK поддерживается системой DB2 (хотя это и не является стандартом языка SQL).

Наконец, необходимо сделать замечание об *эскалации блокировок*, которая реализована во многих системах и представляет собой попытку балансирования между вступающими в конфликт требованиями высокого параллелизма и низкими накладными расходами на управление блокировками. Основная идея заключается в том, что, когда система достигает некоторого заранее заданного порога, она автоматически заменяет множество мелких блокировок одной крупной блокировкой. Например, набор отдельных S-блокировок на уровне кортежа, составляющий IS-блокировку на уровне отношения, можно преобразовать в S-блокировку на уровне отношения. На практике это решение часто оказывается весьма полезным.

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

Предикатные синхронизационные захваты

Рассмотрим предикатные синхронизационные захваты на примере кортежей-"фантомов". Проблема кортежей-"фантомов" относится к более тонким проблемам изолированности транзакций. Она вызывает ситуации, которые также противоречат изолированности пользователей. Рассмотрим следующий сценарий. Транзакция 1

выполняет оператор А 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 А, 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. Поясним как это можно делать.

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

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

имя-атрибута { = > < } значение

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

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации. Пусть R отношение с атрибутами a1, a2, ..., an, а m1, m2, ..., mn - множества допустимых значений a1, a2, ..., an соответственно (все эти множества - конечные). Тогда можно сопоставить R конечное n-мерное пространство возможных значений кортежей R. Любое простое условие "вырезает" m-мерный прямоугольник в этом пространстве (m <= n).

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

Это иллюстрируется следующим примером, показывающим, что в каких бы режимах не требовала транзакция 1 захвата условия (1<=a<=4) & (b=5), а транзакция 2 - условия (1<=a<=5) & (1<=b<=3), эти захваты всегда совместимы.

Пример: (n = 2)

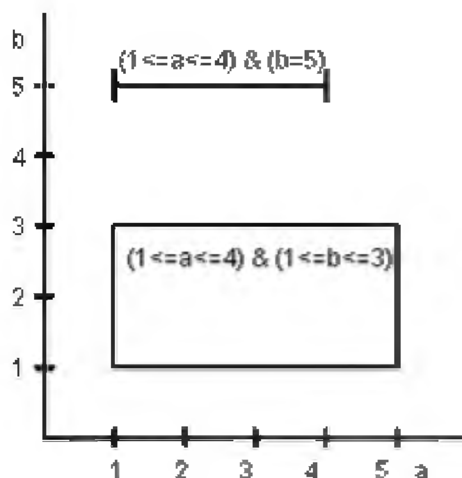


Рис.5.13.

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

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

Вот простой пример возникновения тупика между транзакциями T_1 и T_2:

- транзакции T_1 и T_2 установили монопольные захваты объектов r_1 и r_2 соответственно;
- после этого T_1 требуется совместный захват r_2, а T_2 - совместный захват r_1;
- ни одна из транзакций не может продолжаться, следовательно, монопольные захваты не будут сняты, а совместные - не будут удовлетворены.

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

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. ***Граф ожидания транзакций*** - это ориентированный двудольный граф, в котором существует два типа вершин - вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. В этом графе существует дуга, ведущая из вершины-транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный захват объекта. В графе существует дуга из вершины-объекта к вершине-транзакции, если транзакция ожидает удовлетворения захвата объекта.

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

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

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

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

Грубо говоря, критерием выбора является стоимость транзакции; жертвой выбирается самая дешевая транзакция. Стоимость транзакции определяется на основе многофакторная оценка, в которую с разными весами входят время выполнения, число накопленных захватов, приоритет.

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

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

Заметим, что в централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в по-настоящему распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

Еще одно замечание. Чтобы минимизировать число конфликтов между транзакциями, в некоторых СУБД (например, в Oracle) используется следующее развитие подхода. Монопольный захват объекта блокирует только изменяющие транзакции. После выполнении операции модификации предыдущая версия объекта остается доступной для чтения в других транзакциях. Кратковременная блокировка чтения требуется только на период фиксации изменяющей транзакции, когда обновленные объекты становятся текущими.

Метод временных меток

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

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T_1 началась раньше транзакции T_2, то система обеспечивает такой режим выполнения, как если бы T_1 была целиком выполнена до начала T_2.

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

Перед выполнением операции над объектом r транзакция T_1 выполняет следующие действия:

Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.

Если транзакция T не завершилась, то T_1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T_1 выполняет свою операцию.

Если операции T_1 и T конфликтуют, то если $t(T) > t(T_1)$ (т.е. транзакция T является более "молодой", чем T), производится откат T и T_1 продолжает работу.

Если же $t(T) < t(T_1)$ (T "старше" T_1), то T_1 получает новую временную метку и начинается заново.

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

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

Лекция 6. Разновидности распределенных систем

Возможны однородные и неоднородные распределенные базы данных. В однородном случае каждая локальная база данных управляется одной и той же СУБД. В неоднородной системе локальные базы данных могут относиться даже к разным моделям данных. Сетевая интеграция неоднородных баз данных - это актуальная, но очень сложная проблема. Многие решения известны на теоретическом уровне, но пока не удается справиться с главной проблемой - недостаточной эффективностью интегрированных систем.

Заметим, что более успешно практически решается промежуточная задача - интеграция неоднородных SQL-ориентированных систем. Понятно, что этому в большой степени способствует стандартизация языка SQL и общее следование производителей СУБД принципам открытых систем. Мы ограничимся рассмотрением проблем интеграции однородных распределенных СУБД на примере гипотетической системы System R*. Но прежде чем перейти к ним рассмотрим внутреннюю организацию многопользовательских реляционных СУБД.

Внутренняя организация реляционных СУБД

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.
- *Избыточность хранения данных* Это свойство необходимо поддерживать для выполнения требования надежного хранения баз данных, что обычно реализуется в виде журнала изменений базы данных и копий 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).

System R и **Ingres** имеют два альтернативных подхода к организации реляционной СУБД с точки разделения функций между различными компонентами. В СУБД System R существует 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).

Хранение отношений

Существуют два принципиальных подхода к физическому хранению отношений. Наиболее распространенным является покортежное хранение отношений (кортеж является единицей физического хранения). Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одного отношения и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа.

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

Поскольку гораздо более распространено хранение по строкам, мы рассмотрим немного более подробно этот способ хранения отношений. Типовой, унаследованной от System R, структурой страницы данных является следующая:

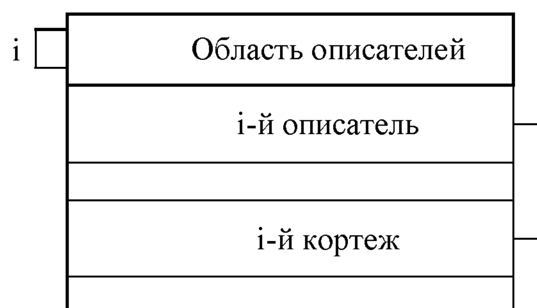


Рис.6.1. Типовая структура страницы

К основным характеристикам этой организации можно отнести следующие:

- Каждый кортеж обладает уникальным идентификатором (tid), не изменяемым во все время существования кортежа. Структура tid следует из приведенного выше рисунка.
- Обычно каждый кортеж хранится целиком в одной странице. Из этого следует, что максимальная длина кортежа любого отношения ограничена размерами страницы. Возникает вопрос: как быть с "длинными" данными, которые в принципе не помещаются в одной странице? Применяются несколько методов. Наиболее простым решением является хранение таких данных в отдельных (вне базы данных) файлах с заменой "длинного" данного в кортеже на имя соответствующего файла. В некоторых системах (например, в предпоследней версии СУБД Informix) такие данные хранились в отдельном наборе страниц внешней памяти, связанном физическими ссылками. Оба эти решения сильно ограничивают возможность работы с длинными данными (как, например, удалить несколько байтов из середины 2-мегабайтной строки?). В настоящее время все чаще используется метод, предложенный несколько лет тому назад в проекте Exodus, когда "длинные" данные организуются в виде B-деревьев последовательностей байтов.
- Как правило, в одной странице данных хранятся кортежи только одного отношения. Существуют, однако, варианты с возможностью хранения в одной странице кортежей нескольких отношений. Это вызывает некоторые дополнительные расходы по части служебной информации (при каждом кортеже нужно хранить информацию о соответствующем отношении), но зато иногда позволяет резко сократить число обменов с внешней памятью при выполнении соединений.
- Изменение схемы хранимого отношения с добавлением нового столбца не вызывает потребности в физической реорганизации отношения. Достаточно лишь изменить информацию в описателе отношения и расширять кортежи только при занесении информации в новый столбец.
- Поскольку отношения могут содержать неопределенные значения, необходима соответствующая поддержка на уровне хранения. Обычно это достигается путем хранения соответствующей шкалы при каждом кортеже, который в принципе может содержать неопределенные значения.
- Проблема распределения памяти в страницах данных связана с проблемами синхронизации и журнализации и не всегда тривиальна. Например, если в ходе выполнения транзакции некоторая страница данных опустошается, то ее нельзя перевести в статус свободных страниц до конца транзакции, поскольку при откате транзакции удаленные при прямом выполнении транзакции и восстановленные при ее откате кортежи должны получить те же самые идентификаторы.
- Распространенным способом повышения эффективности СУБД является кластеризация отношения по значениям одного или нескольких столбцов. Полезным для оптимизации соединений является совместная кластеризация нескольких отношений.
- С целью использования возможностей распараллеливания обменов с внешней памятью иногда применяют схему декластеризованного хранения отношений: кортежи с общим значением столбца декластеризации размещают на разных дисковых устройствах, обмены с которыми можно выполнять в параллель.

Что же касается хранения отношения по столбцам, то основная идея состоит в совместном хранении всех значений одного (или нескольких) столбцов. Для каждого

кортежа отношения хранится кортеж той же степени, состоящий из ссылок на места расположения соответствующих значений столбцов. В последней лекции мы будем рассматривать особенности организации распределенных реляционных СУБД. Одним из приемов является так называемое вертикальное разделение отношений, когда в разных узлах сети хранятся разные проекции данного отношения. Хранение отношения по столбцам в некотором смысле является предельным случаем вертикального разделения отношений.

Индексы

Как бы не были организованы индексы в конкретной СУБД, их основное назначение состоит в обеспечении эффективного прямого доступа к кортежу отношения по ключу. Обычно индекс определяется для одного отношения, и ключом является значение атрибута (возможно, составного). Если ключом индекса является возможный ключ отношения, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа отношения автоматически заводится уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа так или иначе требуется индексная поддержка.

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

Наконец, одним из способов оптимизации выполнения эквисоединения отношений (наиболее распространенная из числа дорогостоящих операций) является организация так называемых мультииндексов для нескольких отношений, обладающих общими атрибутами. Любой из этих атрибутов (или их набор) может выступать в качестве ключа мультииндекса. Значению ключа сопоставляется набор кортежей всех связанных мультииндексом отношений, значения выделенных атрибутов которых совпадают со значением ключа.

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

B-деревья

Видимо, наиболее популярным подходом к организации индексов в базах данных является использование техники B-деревьев. С точки зрения внешнего логического представления B-дерево - это сбалансированное сильно ветвистое дерево во внешней памяти. Сбалансированность означает, что длина пути от корня дерева к любому его листу одна и та же. Ветвистость дерева - это свойство каждого узла дерева ссылаться но большое число узлов-потомков. С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

В типовом случае структура внутренней страницы выглядит следующим образом:

$$N_1 \text{ ключ}(1)\ N_2 \text{ ключ}(2) \ldots N_n \text{ ключ}(n)\ N_{n+1} \text{ ключ}(n+1)$$

При этом выдерживаются следующие свойства:

- ключ(1) <= ключ(2) <= ... <= ключ(n);
- в странице дерева Nm находятся ключи k со значениями ключ(m) <= k <= ключ(m+1).

Листовая страница обычно имеет следующую структуру:

ключ(1) сп(1) ключ(2) сп(2) … ключ(n) сп(n) ключ(t) сп(t)

Листовая страница обладает следующими свойствами:

ключ(1) < ключ(2) < ... < ключ(t);

где сп(r) - упорядоченный список идентификаторов кортежей (tid), включающих значение ключ(r); листовые страницы связаны одно- или двунаправленным списком.

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

Основной "изюминкой" B-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций занесения и удаления записей.

При занесение новой записи выполняется следующим образом:

- *Поиск листовой страницы*. Фактически, производится обычный поиск по ключу. Если в B-дереве не содержится ключ с заданным значением, то будет получен номер страницы, в которой ему надлежит содержаться, и соответствующие координаты внутри страницы.
- *Помещение записи на место*. Естественно, что вся работа производится в буферах оперативной памяти. Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер страницы внешней памяти.
- *Если* после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память, или временно сохранен в оперативной памяти в зависимости от политики управления буферами.
- *Если* же возникло *переполнение буфера* (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы. Для этого запрашивается новая страница внешней памяти, используемая часть буфера разбивается грубо говоря пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенную страницу, а в старой странице модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых страниц.

Чтобы обеспечить доступ от корня дерева к заново заведенной страницы, необходимо соответствующим образом модифицировать внутреннюю страницу, являющуюся предком ранее существовавшей листовой страницы, т.е. вставить в нее соответствующее значение ключа и ссылку на новую страницу. При выполнении этого действия может снова произойти переполнение теперь уже внутренней страницы, и она будет расщеплена на две. В результате потребуется вставить значение ключа и ссылку на новую страницу во внутреннюю страницу-предка выше по иерархии и т.д.

Предельным случаем является переполнение корневой страницы В-дерева. В этом случае она тоже расщепляется на две, и заводится новая корневая страница дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

- *Поиск записи по ключу*. Если запись не найдена, то значит удалять ничего не нужно.
- *Реальное удаление записи в буфере*, в который прочитана соответствующая листовая страница.
- *Если* после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых страницах, являющихся левым или правым братом данной страницы, больше, чем размер страницы, операция завершается.
- *Иначе* производится слияние с правым или левым братом, т.е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и ее левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц.

Чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы - ее предка. При этом может возникнуть потребность в слиянии этой страницы с ее левым или правыми братьями и т.д.

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

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

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

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

Следует заметить, что при организации мультидоступа к В-деревьям, характерного при их использовании в СУБД, приходится решать ряд нетривиальных проблем. Конечно, грубые решения очевидны, например монопольный захват В-дерева на все выполнение операции модификации. Но существуют и более тонкие решения, рассмотрение которых выходит за пределы нашего курса.

И последнее замечание относительно В-деревьев. В литературе вид рассмотренных нами деревьев принято называть B* или B+-деревьями.

Хэширование

Альтернативным и все более популярным подходом к организации индексов является использование техники хэширования. Это очень обширная тема, которая заслуживает отдельного рассмотрения. Мы ограничимся лишь несколькими замечаниями. Общей идеей методов хэширования является применение к значению ключа некоторой

функции свертки (хэш-функции), вырабатывающей значение меньшего размера. Свертка значения ключа затем используется для доступа к записи.

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

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

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

Журнальная информация

Структура журнала обычно является сугубо частным делом конкретной реализации. Мы отметим только самые общие свойства.

Журнал обычно представляет собой чисто последовательный файл с записями переменного размера, которые можно просматривать в прямом или обратном порядке. Обмены производятся стандартными порциями (страницами) с использованием буфера оперативной памяти. В грамотно организованных системах структура (и тем более, смысл) журнальных записей известна только компонентам СУБД, ответственным за журнализацию и восстановление. Поскольку содержимое журнала является критичным при восстановлении базы данных после сбоев, к ведению файла журнала предъявляются особые требования по части надежности. В частности, обычно стремятся поддерживать две идентичные копии журнала на разных устройствах внешней памяти.

Распределенная система управления базами данных System R*

Основную цель проекта можно сформулировать следующим образом: обеспечить средства интеграции локальных баз данных System R, располагающихся в узлах вычислительной сети, с тем, чтобы пользователь, работающий в любом узле сети, имел доступ ко всем этим базам данных так, как если бы они были централизованы. При этом должны обеспечиваться:

- легкость использования системы;
- возможности автономного функционирования при нарушениях связности сети или при административных потребностях;
- высокая степень эффективности.

Для решения этих проблем было необходимо принять ряд проектных решений, касающихся декомпозиции исходного запроса, оптимального выбора способа выполнения запроса, согласованного выполнения транзакций, обеспечения синхронизации, обнаружения и разрешения распределенных тупиков, восстановления состояния баз данных после разного рода сбоев узлов сети.

Легкость использования системы достигается за счет того, что пользователи System R* (разработчики прикладных программ и конечные пользователи) остаются в среде

языка SQL, т.е. могут продолжать работать в тех же внешних условиях, что и в System R (и SQL/DS и DB2). Возможность использования SQL основывается на обеспечении System R* прозрачности местоположения данных. Система автоматически обнаруживает текущее местоположение упоминаемых в запросе пользователя объектов данных; одна и та же прикладная программа, включающая предложения SQL, может быть выполнена в разных узлах сети. При этом в каждом узле сети на этапе компиляции запроса выбирается наиболее оптимальный план выполнения запроса в соответствии с расположением данных в распределенной системе.

Обеспечению автономности узлов сети в System R* уделяется очень большое внимание. Каждая локальная база данных администрируется независимо от других. Возможны автономное подключение новых пользователей, смена версии автономной части системы и т.д. Система спроектирована таким образом, что в ней не требуются централизованные службы именования объектов или обнаружения тупиков. В индивидуальных узлах не требуется наличие глобального знания об операциях, выполняющихся в других узлах сети; работа с доступными базами данных может продолжаться при выходе из строя отдельных узлов сети или линий связи.

Высокая степень эффективности системы является одним из наиболее ключевых требований к распределенным системам управления базами данных вообще и к System R* в частности. Для достижения этой цели используются два основных приема.

Во-первых, в System R* выполнению запроса предшествует его компиляция. В ходе этого процесса производится:

- поиск употребляемых в запросе имен объектов баз данных в распределенном каталоге и замена имен на внутренние идентификаторы;
- проверка прав доступа пользователя, от имени которого производится компиляция, на выполнение соответствующих операций над базами данных и выбор наиболее оптимального глобального плана выполнения запроса, который затем подвергается декомпозиции и по частям рассылается в соответствующие узлы сети, где производится выбор оптимальных локальных планов выполнения компонентов запроса и происходит генерация модулей доступа в машинных кодах.

В результате множество действий производится на стадии компиляции до реального выполнения запроса. Обработанная посредством прекомпилятора System R* прикладная программа, включающая предложения SQL, может в дальнейшем выполняться много раз без дополнительных накладных расходов. Использование распределенного каталога, распределенная компиляция и оптимизация запросов являются наиболее интересными и оригинальными аспектами проекта System R*.

Вторым средством повышения эффективности системы является возможность перемещения удаленных отношений в локальную базу данных. Диалект SQL, используемый в System R*, включает предложение MIGRATE TABLE, при выполнении которого указанное отношение переносится в локальную базу данных. Это средство, находящееся в распоряжении пользователей, конечно, в ряде случаев может помочь добиться более эффективного прохождения транзакций. Естественно, как и для всех операций, операция MIGRATE по отношению к указанному отношению доступна не любому пользователю, а лишь тем, которые обладают соответствующим правом.

Прежде, чем перейти к более детальному изложению наиболее интересных аспектов реализации System R*, упомянем некоторые средства, которые разработчики этой системы предполагали реализовать на начальной стадии проекта, но которые реализованы не были (причем некоторые из них, видимо, и не будут никогда реализованы). Предполагалось иметь в системе средства горизонтального и вертикального разделения отношений распределенной базы данных, средства дублирования отношений в нескольких

узлах с поддержкой согласованности копий и средства поддержания мгновенных снимков состояния баз данных в соответствии с заданным запросом.

Для задания горизонтального разделения отношений (горизонтальная фрагментация) в SQL была введена конструкция вида

```
DISTRIBUTE TABLE <table-name> HORIZONTALLY INTO
   <name> WHERE <predicate> IN SEGMENT &lt;segment-name site&gt;
        ...
   <name> WHERE <predicate> IN SEGMENT &lt;segment-name site&gt;
```

При выполнении предложения такого типа указанное отношение разбивалось на ряд подотношений, содержащих кортежи, удовлетворяющие соответствующему предикату из раздела WHERE, и каждое полученное таким образом подотношение посылалось в указанный узел для хранения в сегменте с указанным именем. Гарантируется согласованное состояние разделов при изменении отношения.

Вертикальное разделение (вертикальная фрагментация) производилось с помощью оператора

```
DISTRIBUTE TABLE <table-name> VERTICALLY INTO
 <name> WHERE <column-name-list> IN SEGMENT &lt;segment-name site&gt;
        ...
 <name> WHERE <column-name-list> IN SEGMENT &lt;segment-name site&gt;
```

При выполнении такого предложения также образовывался набор подотношений с помощью проекции заданного отношения на атрибуты из заданного списка. Каждое полученное подотношение затем посылалось для хранения в сегменте с указанным именем в соответствующий узел. После этого система ответственна за поддержание согласованного состояния образованных разделов.

Горизонтальное и вертикальное разделение отношений реально не используются в System R*, хотя очевидно, что выполнение собственно оператора DISTRIBUTE никаких технических трудностей не вызывает. Трудности возникают при обеспечении согласованности разделов (смотри ниже). Кроме того, разделенные отношения очень трудно использовать. В соответствии с идеологией системы учет наличия разделов отношения в разных узлах сети должен производить оптимизатор, т.е. количество потенциально возможных планов выполнения запросов, которые должны оцениваться оптимизатором, еще более возрастает. При том, что в распределенной системе число возможных планов и так очень велико, и оптимизатор работает на пределе сложности, разумным образом использовать разделенные отношения невозможно. Разработчики оптимизатора System R* не были в состоянии учитывать разделенность отношений. Поэтому и вводить в систему разделенные отношения пока бессмысленно.

Для задания требования поддержки копий отношения в нескольких узлах сети предлагалось использовать новую конструкцию SQL

```
DISTRIBUTE TABLE <table-name> REPLICATED INTO
  <name> IN SEGMENT &lt;segment-name site&gt;
          ...
  <name> IN SEGMENT &lt;segment-name site&gt;
```

При выполнении такого предложения должна была производиться рассылка копий указанного отношения для хранения в именованных сегментах указанных узлов сети. Система должна автоматически поддерживать согласованность копий.

Как и в случае разделенных отношений, кроме существенных проблем поддержания согласованности копий, проблемой является и разумное использование копий, наличие которых должно было бы учитываться оптимизатором.

Создание мгновенного снимка состояния баз данных в соответствии с заданным запросом на выборку должно было производиться с использованием новой конструкции SQL.

```
DEFINE SNAPSHOT <snapshot-name> (<attribute-list>)
   AS <query>
   REFRESHED EVERY <period>
```

При выполнении предложения фактически производится выполнение указанного в нем запроса на выборку, а результирующее отношение сохраняется под указанным в предложении именем в локальной базе данных в том узле, в котором выполняется предложение. После этого мгновенный снимок периодически обновляется в соответствии с запомненным запросом.

Можно обновить мгновенный снимок, не дожидаясь истечения временного интервала, указанного в определении, путем выполнения предложения REFRESH SNAPSHOT <snapshot-name>.

Разумное использование мгновенных снимков более реально, чем использование разделенных отношений и копированных отношений, поскольку их можно в некотором смысле рассматривать как материализованные представления базы данных. Имя мгновенного снимка можно было бы использовать прямо в запросе на выборку там, где можно использовать имена базовых отношений или представлений. Большие проблемы связаны с обновлением отношений через их мгновенные снимки, поскольку в момент обновления содержимое мгновенного снимка может расходиться с текущим содержимым базового отношения.

По отношению к мгновенным снимкам проблем поддержания согласованного состояния мгновенного снимка и базовых отношений не существует, поскольку автоматическое согласование не требуется. Что же касается разделенных отношений и раскопированных отношений, то для них эта проблема общая и достаточно трудная. Во-первых, согласование разделов и копий вызывает существенные накладные расходы при выполнении операций модификации хранимых отношений. Для этого требуется выработка и соблюдение специальных протоколов модификации.

Во-вторых, введение копированных отношений обычно производится не столько для увеличения эффективности системы, сколько для увеличения доступности данных при нарушении связности сети. В системах, в которых применяется этот подход, при нарушении связности сети работа с распределенной базой данных обычно продолжается только в одной из образовавшихся подсетей. При этом для выбора подсети используются алгоритмы голосования; решение принимается на основе учета количества связных узлов сети. Применяются и другие подходы, но все они очень дорогостоящие, а самое главное, они плохо согласуются с базовым подходом System R* по поводу выбора способа выполнения запроса на стадии его компиляции. Поэтому, как нам кажется, в System R* никогда не будут реализованы средства, позволяющие тем или иным способом поддерживать копии отношений в нескольких узлах сети.

Далее мы рассмотрим аспекты проекта System R*, которые нашли отражение в ее реализации и являются на наш взгляд наиболее интересными: средства именования объектов и организацию распределенного каталога баз данных; подход к распределенным компиляции и выполнению запросов; особенности использования представлений; средства оптимизации запросов; особенности управления транзакциями; средства синхронизации и распределенный алгоритм обнаружения синхронизационных тупиков.

Именование объектов и организация распределенного каталога

Напомним прежде всего, что полное имя отношения (базового или представления) в базе данных System R имеет вид имя-пользователя.имя-отношения, где имя-пользователя идентифицирует пользователя - создателя отношения, а имя-отношения - это то имя, которое было указано в предложениях CREATE TABLE или CREATE VIEW. В запросах можно указывать либо это полное имя отношения, либо его локальное имя. Во втором случае при компиляции используются стандартные правила дополнения локального

имени до полного с использованием в качестве составляющей имя-пользователя идентификатора пользователя, от имени которого выполняется компиляция.

В System R* используется развитие этого подхода. Системное имя отношения включает четыре компонента: идентификатор пользователя-создателя отношения; идентификатор узла сети, в котором выполнялась операция создания отношения; локальное имя отношения, присвоенное ему при создании; идентификатор узла, в котором отношение располагалось непосредственно после своего создания (напомним, что отношение может перемещаться из одного узла в другой при выполнении операции MIGRATE).

В запросе на SQL можно использовать системные имена объектов, но разрешается использовать и короткие локальные имена (либо локальное имя, квалифицированное именем пользователя). При этом возможны две интерпретации локального имени. Оно может интерпретироваться как часть системного имени, и в этом случае по умолчанию дополняется до системного, исходя из идентификатора узла, в котором производится компиляция, и идентификатора пользователя, от имени которого она производится (если имя пользователя не указано явно). Вторая возможная интерпретация локального имени заключается в рассмотрении его как имени ранее определенного синонима системного имени.

Для определения синонимов SQL расширен оператором вида

```
DEFINE SYNONYM <relation-name> AS <system-wide-name>.
```

При выполнении такого предложения в локальный каталог заносится соответствующая информация.

Таким образом, при компиляции запроса всегда можно определить системные имена всех употребляемых в нем отношений: либо они явно указаны, либо могут быть получены на основе информации из локальных отношений-каталогов.

Концепция распределенного каталога System R* основана на наличии у каждого объекта распределенной базы данных уникального системного имени. Принято следующее соглашение: информация о размещении любого объекта базы данных (идентификатор текущего узла, в котором размещен объект) сохраняется в локальном каталоге того узла, в котором объект располагался непосредственно после создания (родового узла).

Следовательно, для получения полной информации об отношении в общем случае необходимо сначала воспользоваться локальным каталогом узла, в котором происходит компиляция, затем обратиться к удаленному каталогу родового узла данного отношения и в заключение воспользоваться каталогом текущего узла. Таким образом, для получения точной системной информации о любом отношении распределенной базы данных может потребоваться самое большее два удаленных доступа к отношениям-каталогам.

Применяется некоторая оптимизация этой процедуры. В локальном каталоге узла могут храниться копии элементов каталога других узлов (своего рода кэш-каталог). Согласованность копий элементов каталога не поддерживается. Эта информация используется на первой стадии компиляции запроса (мы рассматриваем распределенную компиляцию в следующем подразделе), а затем, на второй стадии, если информация, касающаяся некоторого объекта, оказалась неточной, она уточняется на основе локального каталога того узла, в котором объект хранится в настоящее время. Обнаружение некорректности копии элемента каталога производится за счет наличия при каждом элементе каталога номера версии. Если учесть достаточную инерционность системной информации, эта оптимизация может оказаться существенной.

Распределенная компиляция запросов

Как мы уже отмечали, запросы на языке SQL до своего реального выполнения подвергаются компиляции. Как и в случае System R компиляция запроса может производиться на стадии прекомпиляции прикладной программы, написанной на

традиционном языке программирования (PL/1, Cobol, ассемблер) с включением предложений SQL, или в динамике выполнения транзакции при выполнении предложения PREPARE. С точки зрения пользователей процесс компиляции в System R* приводит к тем же результатам, что и в System R: для каждого предложения SQL образуется программа к машинных кодах (секция модуля доступа), вызовы которой помещаются в текст исходной прикладной программы.

Однако, в действительности процесс компиляции запроса в System R* намного более сложен, чем в System R, что и естественно по причине гораздо более сложных сетевых взаимодействий, которые потребуются при реальном выполнении транзакции. Распределенная компиляция запросов в System R* включает множество технических ухищрений и тонкостей. Мы не будем касаться их всех в этой статье по причинам недостатка информации и ограниченности объема. Рассмотрим только общую схему распределенной компиляции.

Будем называть главным узлом тот узел сети, в котором инициирован процесс компиляции предложения SQL, и дополнительными узлами - те узлы, которые вовлекаются в этот процесс в ходе его выполнения. На самом грубом уровне процесс компиляции можно разбить на следующие фазы:

1. В главном узле производится грамматический разбор предложения SQL с построением внутреннего представления запроса в виде дерева. На основе информации из локального каталога главного узла и удаленных каталогов дополнительных узлов производится замена имен объектов, фигурирующих в запросе, на их системные идентификаторы.
2. В главном узле генерируется глобальный план выполнения запроса, в котором учитывается лишь порядок взаимодействий узлов при реальном выполнении запроса. Для выработки глобального плана используется расширение техники оптимизации, применяемой в System R. Глобальный план отображается в преобразованном соответствующим образом дереве запроса.
3. Если в глобальном плане выполнения запроса участвуют дополнительные узлы, производится его декомпозиция на части, каждую из которых можно выполнить в одном узле (например, локальная фильтрация отношения в соответствии с заданным в условии выборки предикате ограничения). Соответствующие части запроса (во внутреннем представлении) рассылаются в дополнительные узлы.
4. В каждом узле, участвующем в глобальном плане выполнения запроса (главном и дополнительных) выполняется завершающая стадия выполнения компиляции. Эта стадия включает, по существу, две последние фазы процесса компиляции запроса в System R: оптимизацию и генерацию машинных кодов. Производится проверка прав пользователя, от имени которого производится компиляция, на выполнение соответствующих действий; происходит обработка представлений базы данных (здесь имеются тонкости, связанные с тем, что представления могут включать удаленные отношения; ниже мы еще остановимся на этом, а пока будем считать, что в запросе употребляются только имена базовых отношений); осуществляется локальная оптимизация обрабатываемой части запроса в соответствии с имеющимися индексами; наконец, производится генерация кода.

Управление транзакциями и синхронизация

Выполнение транзакции в распределенной системе управления базами данных System R*, естественно, является распределенным. Транзакция начинается в главном узле при обращении к какой-либо секции ранее подготовленного (на этапе компиляции) модуля доступа. Как и в System R, модуль доступа загружается в виртуальную память задачи, обращение к секции модуля доступа - это вызов подпрограммы. Однако, в отличие от System R, эта подпрограмма, кроме своего локального программного кода и вызовов функций RSS, содержит еще и вызовы удаленных подсекций модуля доступа. Эти вызовы

интерпретируются в духе вызовов удаленных процедур. Тем самым выполнение одной транзакции, инициированной в некотором узле сети А влечет, вообще говоря, инициирование транзакций в дополнительных узлах. Основной новой по сравнению с System R проблемой является проблема согласованного завершения распределенной транзакции, чтобы результаты ее выполнения во всех затронутых ею узлах были либо отображены в состояние локальных баз данных, либо полностью отсутствовали.

Для достижения этой цели в System R* используется двухфазный протокол завершения распределенной транзакции. Этот протокол является общеупотребимым в распределенных системах баз данных и описан во многих литературных источниках. Поэтому мы здесь опишем его очень кратко и неформально.

Для описания протокола используется следующая модель. Имеется ряд независимых транзакций-участников распределенной транзакции, выполняющихся под управлением транзакции-координатора. Решение об окончании распределенной транзакции принимается координатором. После этого выполняется первая фаза завершения транзакции, когда координатор передает каждому из участников сообщение "подготовиться к завершению". Получив такое сообщение, каждый участник переходит в состояние готовности как к немедленному завершению транзакции, так и к ее откату. В терминах System R* это означает, что буфер журнала с записями об изменениях базы данных участника выталкиваются на внешнюю память, но синхронизационные захваты не снимаются. После этого каждый участник, успешно выполнивший подготовительные действия, посылает координатору сообщение "готов к завершению". Если координатор получает такие сообщения ото всех участников, то он начинает вторую фазу завершения, рассылая всем участникам сообщение "завершить транзакцию", и это считается завершением распределенной транзакции. Если не все участники успешно выполнили первую фазу, то координатор рассылает всем участникам сообщение "откатить транзакцию", и тогда эффект воздействия распределенной транзакции на состояние баз данных отсутствует.

По отношению к особенностям реализации двухфазного протокола завершения транзакции в System R* заметим еще следующее. В качестве координатора выступает транзакция, выполняющаяся в главном узле, т.е. та, по инициативе которой возникли дополнительные транзакции. Тем самым, наличие центрального координирующего узла не требуется, что соответствует требованию автономности узлов. Для откатов транзакций используется базовый механизм точек сохранения System R. Наконец, классический протокол двухфазного завершения оптимизирован, чтобы сократить число необходимых сообщений.

Как и в System R, согласованность состояния баз данных при параллельном выполнении нескольких транзакций в System R* обеспечивается на основе механизма синхронизационных захватов объектов базы данных при соблюдении двухфазного протокола захватов. Напомним, что это означает разбиение каждой транзакции с точки зрения синхронизации на две фазы - рабочую фазу, на которой захваты только устанавливаются, и фазу завершения, когда все захваты объектов базы данных, произведенные данной транзакцией, снимаются. Синхронизация производится в точности так же, как и в System R: каждая транзакция-участник обращается к локальной базе данных через RSS своего узла. Основной новой проблемой является проблема возможных распределенных тупиков, которые могут возникнуть между несколькими распределенными транзакциями, выполняющимися параллельно. (Тупики между транзакциями - участниками одной распределенной транзакции невозможны, поскольку все участники получают один общий идентификатор транзакции и не конфликтуют по синхронизации). Для обнаружения распределенных синхронизационных тупиков в System R* применяется оригинальный распределенный алгоритм, не нарушающий требования автономности узлов сети и минимизирующий число передаваемых по сети сообщений и необходимую процессорную обработку.

Основная идея алгоритма состоит в том, что в каждом узле периодически производится анализ на предмет существования тупика с использованием информации о связях транзакций по ожиданию ресурсов, локальной в данном узле и полученной от других узлов. При проведении этого анализа обнаруживаются либо циклы ожиданий, что означает наличие тупика, либо потенциальные циклы, которые необходимо уточнить в других узлах. Эти потенциальные циклы представляются в виде специального вида строк. Строка представляет собой по сути дела список транзакций. Все транзакции упорядочены в соответствии со значениями своих идентификаторов ("номеров транзакций"). Строка передается для дальнейшего анализа в следующий узел (узел, в котором выполняется самая правая в строке транзакция) только в том случае, если номер первой транзакции в строке меньше номера последней транзакции. (Это оптимизация, уменьшающая число передаваемых по сети сообщений). Этот процесс продолжается до обнаружения тупика.

Если обнаруживается наличие синхронизационного тупика, он разрушается за счет уничтожения (отката) одной из транзакций, входящей в цикл. В качестве жертвы выбирается транзакция, выполнившая к этому моменту наименьший объем работы. Эта информация также передается по сети вместе со строками, описывающими связи транзакций по ожиданию.

Интегрированные или федеративные системы и мультибазы данных

Направление интегрированных или федеративных систем неоднородных БД и мульти-БД появилось в связи с необходимостью комплексирования систем БД, основанных на разных моделях данных и управляемых разными СУБД.

Основной задачей интеграции неоднородных БД является предоставление пользователям интегрированной системы глобальной схемы БД, представленной в некоторой модели данных, и автоматическое преобразование операторов манипулирования БД глобального уровня в операторы, понятные соответствующим локальным СУБД. В теоретическом плане проблемы преобразования решены, имеются реализации.

При строгой интеграции неоднородных БД локальные системы БД утрачивают свою автономность. После включения локальной БД в федеративную систему все дальнейшие действия с ней, включая администрирование, должны вестись на глобальном уровне. Поскольку пользователи часто не соглашаются утрачивать локальную автономность, желая тем не менее иметь возможность работать со всеми локальными СУБД на одном языке и формулировать запросы с одновременным указанием разных локальных БД, развивается направление мульти-БД. В системах мульти-БД не поддерживается глобальная схема интегрированной БД и применяются специальные способы именования для доступа к объектам локальных БД. Как правило, в таких системах на глобальном уровне допускается только выборка данных. Это позволяет сохранить автономность локальных БД.

Как правило, интегрировать приходится неоднородные БД, распределенные в вычислительной сети. Это в значительной степени усложняет реализацию. Дополнительно к собственным проблемам интеграции приходится решать все проблемы, присущие распределенным СУБД: управление глобальными транзакциями, сетевую оптимизацию запросов и т.д. Очень трудно добиться эффективности.

Как правило, для внешнего представления интегрированных и мульти-БД используется (иногда расширенная) реляционная модель данных. В последнее время все чаще предлагается использовать объектно-ориентированные модели, но на практике пока основой является реляционная модель. Поэтому, в частности, включение в интегрированную систему локальной реляционной СУБД существенно проще и эффективнее, чем включение СУБД, основанной на другой модели данных.

Системы управления базами данных следующего поколения

Лекция 7. Современные направления исследований и разработок

Несмотря на свою их привлекательность, классические реляционные системы управления базами данных являются ограниченными. Они идеально походят для таких традиционных приложений, как системы резервирования билетов или мест в гостиницах, а также банковских систем, но их применение в системах автоматизации проектирования, интеллектуальных системах обучения и других системах, основанных на знаниях, часто является затруднительным. Это прежде всего связано с примитивностью структур данных, лежащих в основе реляционной модели данных. Плоские нормализованные отношения универсальны и теоретически достаточны для представления данных любой предметной области. Однако в нетрадиционных приложениях в базе данных появляются сотни, если не тысячи таблиц, над которыми постоянно выполняются дорогостоящие операции соединения, необходимые для воссоздания сложных структур данных, присущих предметной области.

Другим серьезным ограничением реляционных систем являются их относительно слабые возможности по части представления семантики приложения. Самое большее, что обеспечивают реляционные СУБД,- это возможность формулирования и поддержки ограничений целостности данных. После проектирования реляционной базы данных многие знания проектировщика остаются зафиксированными в лучшем случае на бумаге по причине отсутствия в системе соответствующих выразительных средств.

Осознавая эти ограничения и недостатки реляционных систем, исследователи в области баз данных выполняют многочисленные проекты, основанные на идеях, выходящих за пределы реляционной модели данных. По всей видимости, какая-либо из этих работ станет основой систем баз данных будущего. Следует заметить, что тематика современных исследований, относящихся к базам данных, исключительно широка. В завершающей первой части курса приведем короткий обзор наиболее важных направлений.

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.

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).

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.

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 баз 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 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 возможным 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 (compiler 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 которой в достаточной степени соответствуют потребностям приложения. На сегодняшний день на коммерческом рынке такие "генерационные" системы отсутствуют (например, при выборе сервера системы Oracle невозможно 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 третьей лекции 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 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 t_1 и уничтоженного в момент времени t_2, в БД сохраняются (и доступны пользователям) все его состояния во временном интервале $[t_1,t_2]$.

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.

Лекция 8. 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, в систематической манере проанализируем 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, 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, относящихся 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 "Object", 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 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 A: 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 О2

Проект О2 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 О2.

Прототип системы функционировал в режиме клиент/сервер в локальной сети рабочих станций SUN с 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 О2 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 О2 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.

Лекция 9. Системы баз данных, основанные на правилах

В этой очень краткой лекции мы рассмотрим последнюю тему этого курса - системы баз данных, основанные на правилах. Более точно можно было бы сказать, что завершающая первую часть курса 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 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 во времена System R. На самом деле, это есть определение триггера или условного воздействия, т.е. 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 интенсиональной 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.

Часть II. Распределенные вычисления

Лекция 1. Общие вопросы организации распределенных вычислений

В данной лекции будут рассмотрены преимущества распределенной обработки по сравнению с традиционными способами построения информационных систем, типы используемых для этой цели сервисов, стандарты и спецификации, используемые при их создании, а также некоторые программные средства для их реализации. Также рассмотрены некоторые примеры реализации таких сервисов, созданные главным образом с помощью средств разработки и технологий Inprise Corporation (Borland).

В общем случае понятие сервиса отнюдь не ограничивается информационной системой какой-либо организации, предоставляющей сотрудникам доступ к корпоративным данным. Сервисом может быть и доступ к тем или иным файлам, хранящимся в локальной сети, и работа с электронной почтой, и доступ в Internet, и использование сетевого принтера или модема, и проведение каких-либо расчетов. Доступность того или иного сервиса в сети нередко определяется тем, какие стандарты он поддерживает (имеются в виду стандартные программные интерфейсы и стандартные протоколы обмена данными).

Современная информационная система состоит, как правило, из стандартного набора программных компонентов и сервисов. Наиболее важным из таких компонентов является собственно база данных, то есть набор файлов, содержащих данные компании. Этот набор файлов может обслуживаться сервисом, называемым сервером баз данных, если СУБД серверная, или файловыми сервисами операционной системы того компьютера, на котором эти файлы расположены, если СУБД не является серверной. Следующим важным компонентом такой системы является набор пользовательских приложений, используемых для редактирования и просмотра данных на рабочих станциях сотрудников. В этом случае говорят, что такие приложения содержат *презентационную логику* информационной системы. Нередко пользовательские приложения используются для проведения других операций с данными (проверка допустимости данных, статистическая обработка, генерация отчетов и др.). В этом случае говорят о том, что такое приложение содержит алгоритмы прикладной обработки данных. Еще один компонент, без которого работа сетевой информационной системы невозможна - это средства обеспечения доступности данных из СУБД в пользовательском приложении. Набор этих средств существенно зависит от того, является ли СУБД серверной. Как минимум, во всех случаях он включает средства сетевого доступа, базирующиеся на сетевых средствах операционных систем, используемых для эксплуатации СУБД и пользовательских приложений. Сетевые средства операционных систем включают, как минимум, поддержку сетевых протоколов, обеспечивающих этот доступ.

В случае серверных СУБД к этому набору добавляются средства взаимодействия пользовательского приложения и сервера баз данных, использующие ту же самую поддержку сетевых протоколов операционными системами. Эти средства обычно включают клиентскую часть серверной СУБД, содержащую, как правило, низкоуровневое API (Application Program Interface - прикладной программный интерфейс) взаимодействия с сервером баз данных. Помимо этого, средства обеспечения доступности данных нередко содержат библиотеки, содержащие высокоуровневые функции доступа к данным. Эти функции упрощают использование клиентской части, если СУБД серверная, либо реализуют стандартные операции с данными, если СУБД не является серверной. В случае пользовательских приложений, созданных с помощью средств разработки Inprise, это библиотека Borland Database Engine (BDE) и драйверы SQL Links, в случае использования

средств разработки Microsoft и многих других производителей (а иногда и средств разработки Inprise) - ODBC-драйверы.

Типичные проблемы эксплуатации информационных систем и способы их решения

Классические многопользовательские системы, как правило, содержат последние два компонента информационной системы на рабочих станциях пользователей. Из этого следует, что подобные рабочие станции должны предоставлять для самих себя весь требующийся для этого набор сервисов и содержать соответствующее программное обеспечение для их функционирования. Подобное требование нередко усложняет технические требования, предъявляемые к аппаратной части клиентской рабочей станции, и в конечном итоге приводит к удорожанию всей системы в целом (рис. 1.1)

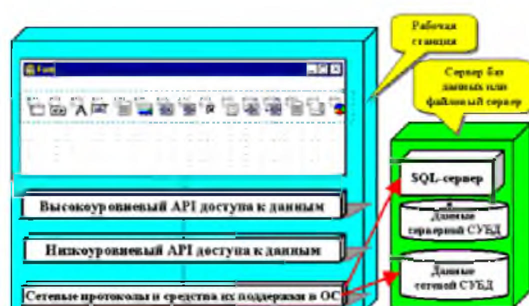


Рис. 1.1. Классическая информационная система

Следует также отметить, что подобное программное обеспечение требует обычно проведения работ по его настройке и поддержанию этих настроек в рабочем состоянии. Так, пользовательское приложение должно, как минимум, "знать" о том, где расположены используемые им данные, какого они типа (имеется в виду тип серверной СУБД либо формат данных сетевой СУБД), с помощью какого сетевого протокола они доступны, каков поддерживаемый базой данных язык, определяющий порядок алфавитной сортировки и индексирования данных. Подобная работа нередко является весьма трудоемким процессом, особенно при большом количестве и неоднородном парке рабочих станций. Отметим, что далеко не все компоненты подобного программного обеспечения могут быть включены в состав дистрибутива пользовательского приложения, так как многие из них являются предметом лицензирования и продажи. Кроме того, чем сложнее конфигурация, обеспечивающая доступ к данным рабочей станции, тем чаще происходят нарушения в ее работе. По данным некоторых западных источников, переконфигурация и сопровождение программного обеспечения, обеспечивающего доступ рабочих станций к данным, приводит в среднем к четырем дням простоя рабочей станции в год.

Для решения этих проблем, в последнее время получает все большее распространение идея создания новых сервисов, общих для пользователей информационной системы. Такие сервисы, как правило, являются *сервисами промежуточного слоя* (middleware services), поскольку занимают промежуточный уровень между данными и сервисами, их обслуживающими, с одной стороны, и пользовательскими приложениями, ориентированными на конкретную предметную область, с другой стороны. Эти сервисы обычно обладают минимальным пользовательским интерфейсом или не имеют его вовсе. Нередко они могут быть реализованы для нескольких различных платформ, так как являются сервисами более высокого уровня, чем сервисы, специфичные для данной операционной системы или СУБД. Такие сервисы могут быть реализованы внутри приложений или библиотек (такие приложения или библиотеки обычно называются *серверами приложений - Application*

Server), а также в виде служб операционных систем. Пользовательские приложения, использующие сервисы промежуточного слоя, обычно называются *клиентами*.

Технологии, используемые для реализации таких сервисов, могут быть различными. В частности, их реализация может использовать технологию и стандарты DCE (Distributed Computing Environment), разработанные OSF (Open Software Foundation), как это сделано в Inprise Entera. Можно реализовать такие сервисы с использованием спецификации CORBA (Common Object Request Broker Architecture), разработанной консорциумом OMG (Object Management Group). В обоих случаях набор возможных клиентских и серверных платформ весьма широк и отнюдь не ограничивается различными версиями Windows. Если же речь идет об относительно недорогих решениях на основе Windows, вполне допустимо использовать DCOM (Distributed Component Object Model) либо различные расширения COM (например, технологию Inprise MIDAS) и реализовывать сервисы middleware внутри серверов автоматизации или компонентов Microsoft Transaction Server .

Сервисы промежуточного слоя и серверы приложений

Одним из наиболее модных на сегодняшний день типов серверов приложений являются *серверы доступа к данным* (Data Access Server), реализуемые, как правило, в виде приложений (реже - в виде библиотек). Они содержат функциональность, связанную с доступом к данным (а нередко и какую-либо иную функциональность, например, статистическую обработку этих данных или генерацию отчетов). Как правило, такие приложения-серверы сами являются клиентами серверных СУБД. В любом случае такие серверы используют перечисленные выше библиотеки доступа к данным. При грамотной организации разделения функций между пользовательским приложением и сервером доступа к данным конфигурация программного обеспечения рабочей станции сводится к информированию пользовательского приложения о том, как называется (или идентифицируется иным способом) нужный ему сервис и на каком компьютере сети должен находиться либо непосредственно он сам, либо некий сервис-посредник, чья задача заключается в поиске нужного сервиса для данного клиента (рис. 1.2).

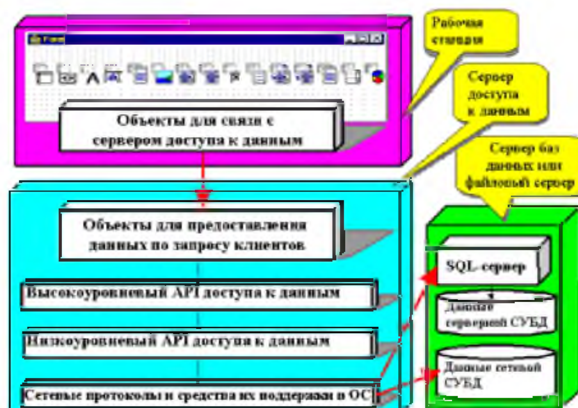


Рис. 1.2. Информационная система с сервером доступа к данным

К этой же категории сервисов можно отнести так называемые *мониторы транзакций*. Мониторы транзакций, как правило, применяются в информационных системах, использующих распределенные базы данных и содержащих приложения, использующие одновременный доступ к нескольким локальным базам данных, включающий распределенные транзакции. Если обычные транзакции внутри одной базы данных с успехом поддерживаются серверными СУБД, лишь бы были реализованы в виде объектов базы данных (индексов, триггеров, хранимых процедур, серверных ограничений и др.) соответствующие правила их выполнения, то распределенные транзакции требуют

отдельной поддержки за пределами СУБД. Именно эта поддержка и реализуется в мониторах транзакций (рис.1.3).

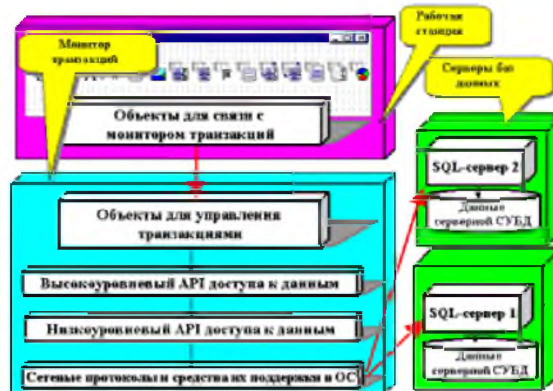


Рис. 1.3. Информационная система с монитором транзакций

Отметим, однако, что в виде отдельного сервиса может быть реализован не только доступ к данным, но и любая другая функциональность пользовательского приложения, например, многомерный анализ и статистическая обработка данных, генерация и печать отчетов, проведение расчетов, обеспечение шифрования данных и многое другое. В этом случае говорят о *серверах функциональности* (functionality server). Отметим, что сервер функциональности - более общее понятие, чем сервер доступа к данным; последний есть лишь частный случай сервера функциональности. Один сервер функциональности может в общем случае предоставлять несколько сервисов. Как правило, на сервер функциональности возлагаются задачи, требующие нестандартных ресурсов:

- избыточного по сравнению с обычной рабочей станцией объема оперативной памяти,
- нестандартного оборудования,
- нестандартной операционной системы или иного программного обеспечения (в том числе, например, библиотек для доступа к данным),
- и др.

Специализированные сервисы

Особым видом сервисов промежуточного слоя являются так называемые специализированные сервисы. Эти сервисы предназначены для обеспечения нормального функционирования системы, содержащей набор серверов функциональности.

К сожалению, терминология (как русская, так и английская), употребляемая при описании таких сервисов, весьма разнообразна и существенно зависит от того, какой спецификации подчиняется их работа и какая реализация данной спецификации используется в конкретном программном продукте, частью которого является данный сервис. Нередко один и тот же термин (например, *брокер, агент*, или *демон*) обозначает различные по своему назначению сервисы в различных спецификациях и различных продуктах, а иногда обозначает не сервис, а просто концепцию, реализованную в сервисах с другими названиями. У некоторых англоязычных терминов имеется несколько разных вариантов русских переводов, некоторые не имеют их вовсе. Поэтому к употребляемой терминологии следует относиться с осторожностью.

Наиболее распространенными из специализированных сервисов являются сервисы, позволяющие на заданных условиях определенным пользователям получить доступ к тому или иному серверу функциональности, содержащемуся на компьютере, где функционирует данный специализированный сервис. Иногда такой сервис может быть выполнен в виде приложения, иногда - в виде сервиса операционной системы. В случае реализации спецификации в продукте Inprise Visibroker он называется *Object Activation Daemon*, в случае использования доступа к COM-серверам с помощью Inprise OLEnterprise

- *Object Factory*, в случае использования доступа к COM-серверам с помощью протокола TCP/IP - *Borland Socket Server*, в случае Inprise AppCenter - *AppCenter Agent*. В случае использования соответствующего сервиса Microsoft DCOM по отношению к нему и другим подобным сервисам иногда употребляется термин *Service Control Manager* .

Действия этого сервиса действительно напоминают действия агента вражеской разведки - доступ к серверам функциональности данного компьютера возможен только в том случае, если такой сервис запущен. Это диктуется обычными соображениями безопасности - было бы неразумным предоставлять возможность кому угодно использовать предоставляемые данным компьютером сервисы в любое время.

Помимо предоставления доступа к серверу функциональности на приложение, содержащее такой сервис, могут возлагаться и другие обязанности (например, запустить сервер функциональности, как это делает Object Activation Daemon, или передавать и принимать данные, как это делает Borland Socket Server).

Еще один распространенный (но не являющийся обязательным) тип специализированных сервисов - это сервисы, занимающиеся поиском серверов функциональности для обратившихся к ним клиентов и выступающие в качестве посредника между клиентом, нуждающимся в том или ином сервисе, и поставщиком сервиса (в данном случае сервером функциональности), сводя их между собой. Иногда такие сервисы называются общим термином *Directory Service*. Обычно такие сервисы используются в системах, содержащих несколько одинаковых серверов функциональности, и подключают обратившихся к ним клиентов к этим серверам в соответствии с установленными для этой системы правилами:

- случайным образом, чем достигается баланс загрузки серверов клиентскими приложениями;
- к какому-либо конкретному серверу, а в случае сбоя - к другому, считающемуся резервным;

Правила выбора сервера для обратившегося клиента могут быть самыми разнообразными.

В случае использования Microsoft DCOM наличие такого сервиса не предполагается, равно как и в случае использования Borland Socket Server. В случае расширения COM с помощью Inprise OLEnterprise этот сервис (в данной реализации он называется *Business Object Broker*) может как использоваться, так и не использоваться. В случае Inprise Entera 3.2 этот сервис используется обязательно (в данной реализации он называется *Entera Broker*). В случае Inprise AppCenter использование подобного сервиса также обязательно (в этом случае он называется *AppCenter Broker*, при этом Entera Broker для него может являться сервером функциональности, который следует искать). А вот в спецификации CORBA и ее реализациях этот сервис называется *Object Agent*.

Рис. 1.4. Система, использующая Directory Service

Отметим, однако, что в некоторых источниках термин *Broker* нередко означает просто некую транспортную службу, обеспечивающую передачу серверу запросов клиента и обмен данными между ними. При этом такая служба может быть реализована внутри какого-либо служебного приложения (как, например, это сделано в Borland Socket Server), а может фактически содержаться внутри самого сервера функциональности и использующего его клиента (так, например, устроены клиенты и серверы Entera; при этом сам термин *broker* в Entera означает Directory Service). В случае же CORBA этот термин обозначает скорее концепцию, нежели конкретный подлежащий реализации сервис.

Лекция 2. Регистрационные базы данных и идентификация серверов и сервисов

Рассмотрим вопрос запуска сервера по запросу клиента. Если реализация сервера одна, и клиенту известно, как ее идентифицировать и на каком компьютере сети ее искать, он обращается к сервису, ответственному за предоставление доступа к данной реализации (Service Control Manager). В случае получения разрешения этот сервис обращается к другому сервису, ответственному за запуск сервера функциональности (как было сказано выше, оба сервиса могут и часто бывают реализованы в одном приложении). Этим другим сервисом либо запускается соответствующее приложение (если сервер еще не запущен, или если каждому клиенту нужен свой экземпляр сервера), либо внутри уже запущенного сервера создаются объекты, взаимодействующие с данным клиентом.

Если реализаций сервиса несколько, удаленный запуск одной из них по запросу клиента может быть осуществлен в том случае, если сведения о местоположении этих реализаций доступны либо самому клиенту, либо сервису, который ищет реализацию по его запросу. В простейшем случае список возможных реализаций просто содержится в клиентском приложении. Например, компонент SimpleObjectBroker в Delphi 4 как раз содержит такой список; клиентское приложение, содержащее такой компонент, будет при каждом запуске случайным образом подключаться к одной из реализаций, указанной в списке. Однако в общем случае, особенно когда для поиска реализации используются специализированные сервисы, такой список хранится отдельно от клиентского приложения.

Естественно, для этого в системе должна существовать некая специализированная база данных, в которой содержатся сведения о сервисах и серверах, содержащих их реализацию. Общей спецификации, которой могло бы подчиняться создание такой базы данных, не существует, так как в общем случае такая база данных может содержаться на любой платформе. В случае COM и его расширений (таких как OLEnterprise) роль такой базы данных с успехом выполняет реестр Windows. В случае CORBA эти сведения хранятся в двух репозитариях (в одном регистрируются интерфейсы сервера, в другом - их реализации, то есть конкретные приложения-серверы). Inprise AppCenter использует свою собственную базу данных (ее реализация существует для нескольких платформ), при этом она управляется специальным приложеним, регистрирующим обращения к ней других сервисов.

Отметим, что удаленный запуск сервера по запросу брокера или клиента осуществляется в том случае, если выполняются условия, при которых он может быть запущен. Как было сказано ранее, на компьютере должен быть запущен сервис, предоставляющий разрешение на удаленный запуск данного сервера , и выполняются условия, при которых это разрешение может быть получено (например, пользователь клиентского приложения имеет право обращаться к этому серверу).

Необходимость регистрации серверов и сервисов зависит от конкретной реализации способа удаленного доступа. COM-сервер автоматически регистрирует себя в реестре Windows после первого запуска, но этот первый запуск должен быть осуществлен не

удаленно, а локально. При этом запись о COM-сервере обязана содержаться в реестре компьютера, содержащего сервер. При использовании DCOM и OLEnterprise удаленный сервер функциональности должен быть зарегистрирован также и в реестре клиентской рабочей станции, при этом использование DCOM предполагает, что для регистрации сервера на рабочей станции нужно просто запустить его на ней. OLEnterprise, в отличие от Microsoft DCOM, предоставляет средства для импорта записи о сервере функциональности из реестра компьютера, содержащего сервер, в реестры рабочих станций. При использовании же Borland Socket Server сведения о сервере обязаны содержаться в клиентском приложении, а регистрация сервера на рабочей станции не требуется.

CORBA-сервер может не быть зарегистрирован в репозитариях, но при этом он не будет запускаться автоматически ни по запросу клиентов, ни по запросу служебных сервисов. Для регистрации интерфейсов и реализаций CORBA-серверов существуют специальные утилиты.

Серверы Entera 3.2 (равно как и другие приложения) можно зарегистрировать в базе данных Inprise AppCenter, а можно не регистрировать нигде. В этом случае сервер Entera 3.2 может быть найден сервисами и клиентами только в том случае, если он уже запущен.

В базе данных Inprise AppCenter можно регистрировать самые разнообразные сервисы, серверы и приложения, а также создавать так называемые конфигурации приложений, описывая правила, которым подчиняется их запуск и остановка. Эти правила могут быть весьма сложными, так как ApCenter, по существу, представляет собой средство управления серверами и сервисами в распределенных системах.

Естественно, если сервер функциональности нигде не зарегистрирован и не запущен в данный момент, а сведения о его возможных реализациях не содержатся в клиентском приложении, то он не будет найден ни служебными сервисами, ни клиентами.

Общих правил для идентификации серверов и содержащиеся в них сервисов на все случаи жизни не существуют. Однако сейчас довольно часто принято присваивать серверам и сервисам уникальные идентификаторы UUID (Universal Unique Identifier), представляющие собой 128-разрядные значения, сгенерированные с помощью алгоритма, определенного OSF (Open System Foundation), на основе IP-адреса компьютера и иных его характеристик, и гарантирующего с высокой вероятностью уникальность сгенерированного значения. Реализация этого алгоритма содержится, например, в функции Windows API CoCreateGUID. В случае COM эти идентификаторы называются GUID (Global Unique Identifier), IID (Interface Identifier), CLSID (Class Identifier). Именно по этим идентификаторам обычно клиенты и другие сервисы в большинстве случаев ищут реализацию нужного сервера (или предоставляемого им сервиса). Тем не менее, в некоторых реализациях можно осуществлять поиск и по другим признакам (имя приложения, имя сервиса и др.). Бывают случаи, когда UUID генерируется с целью соответствия стандартам, но в действительности не используется (например, при использовании серверов Entera 3.2 и доступа к ним непосредственно с помощью протокола TCP/IP).

Схема взаимодействия клиентов и серверов

Создание сервисов и серверов middleware имеет смысл главным образом в том случае, когда клиенты и сервер функционируют на *разных* компьютерах. Рассмотрим способы обращения клиента к объектам, содержащимся в оперативной памяти другого компьютера (и в общем случае созданным в другой операционной системе). В настоящее время существует немало способов реализации подобного взаимодействия. Однако, эти способы базируются на одной и той же придуманной много лет назад идее осуществления вызовов удаленных процедур путем передачи данных между объектами внутри клиента и внутри сервера (рис. 2.1).

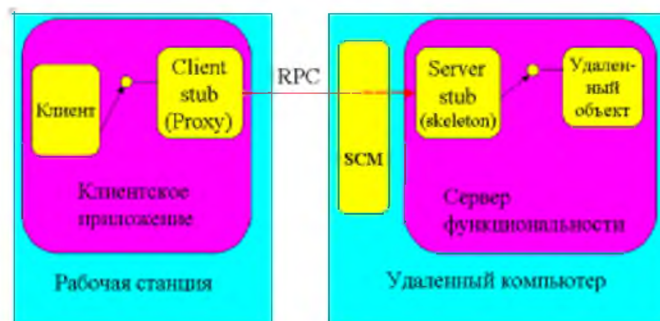


Рис. 2.1. Осуществление вызовов удаленных процедур

При обращении клиента к удаленному серверу функциональности внутри адресного пространства сервера создается так называемый серверный stub-объект (в терминологии CORBA он называется skeleton, в терминологии COM – stub. Некоторые авторы переводят этот термин как "заглушка". Этот объект является представителем данного клиента в адресном пространстве сервера (при многопользовательском доступе к одному и тому же экземпляру сервера этих объектов создается несколько).

Внутри адресного пространства клиента создается клиентский stub-объект (в терминологии CORBA он называется stub, в терминологии COM – proxy. Иногда этот термин переводят как "заместитель". Он может обладать таким же списком методов, как соответствующий ему серверный stub-объект, но никогда не содержит их истинной реализации. В лучшем случае вместо истинной реализации могут присутствовать функции API из библиотеки, реализующей вызовы удаленных процедур и передачу данных с помощью того или иного сетевого протокола. Так происходит в случае stub-кода, сгенерированного утилитами из комплекта поставки Inprise Entera. Эти два stub-объекта общаются между собой, обмениваясь данными примерно так же, как взаимодействуют между собой сетевые приложения и сервисы. Данные, передаваемые передающим stub-объектом, формируются в пакеты данных, которые с помощью сетевых протоколов передаются принимающему stub-объекту (этот процесс иногда называется *маршалингом* или *маршрутизацией*). Принимающий stub-объект расшифровывает пакеты данных.

При отсоединении клиента от сервера соответствующая пара stub-объектов уничтожается. При этом в том случае, когда первый из подключившихся к серверу клиентов инициировал сам или с помощью служебного сервера запуск сервера, при отключении всех клиентов сервер, как правило, должен прекратить свою работу. Если же сервер был запущен пользователем вручную, как правило, отсутствие подключенных клиентов обычно не является основанием для прекращения его работы.

Многие инструменты для создания серверов функциональности содержат в комплекте поставки утилиты для автоматической генерации stub-кода. Код генерируется на основании описаний интерфейса сервера. Во многих случаях эти описания создаются на языке IDL (Interface Definition Language).

Отметим, что существует несколько диалектов IDL (для COM, для CORBA и др.). Тем не менее, различия между ними невелики. Delphi 4 поддерживает как COM IDL, так и CORBA IDL.

Сервер приложений Inprise Entera содержит специальный кодогенератор для большого количества языков программирования. Существует компилятор MIDL для генерации stub-кода и proxy-кода на C++ на основе COM IDL. Он активно используется при создании серверов и клиентов с помощью Microsoft Visual C++. В случае CORBA также имеются кодогенераторы для C++ и Java. При создании серверов функциональности, реализованных в виде серверов автоматизации, с помощью Delphi и C++Builder, stub-код генерируется автоматически при создании соответствующих классов, в этом случае создавать описания с помощью IDL нет необходимости. Однако в этом случае всегда можно сгенерировать описание на IDL (как для COM, так и для CORBA) на основании созданной библиотеки типов сервера. Фактически IDL - это стандарт,

позволяющий описывать вызываемые методы сервера и их параметры, не вдаваясь в детали и правила реализации серверов и клиентов на том или ином языке программирования. Используя IDL, можно описать интерфейсы сервера, а затем создать его реализацию на любом языке программирования с помощью широкого спектра средств разработки для различных платформ, равно как и реализацию клиента. В определенном смысле IDL - это стандарт для описания взаимодействия между компонентами распределенной системы, не зависящий от деталей реализации, языков программирования и платформ.

Лекция 3. Использование Microsoft Transaction Server для управления распределенными транзакциями

COM и распределенные вычисления

Выше во второй части курса были рассмотрены общие вопросы организации распределенных вычислений и общие принципы взаимодействия клиентов и серверов в распределенных системах. Данная лекция посвящена одной из многочисленных реализаций технологии распределенных вычислений - технологии Microsoft COM (точнее, ее расширению - COM+).

В отличие от технологий CORBA (Component Object Request Broker Architecture) или DCE (Distributed Computing Environment), появившихся изначально в виде спецификаций и лишь затем - в виде конкретных реализаций различных производителей, Microsoft COM появилась одновременно и в виде спецификации (т.е. правил создания серверов и клиентов, описания соответствующего API, диалекта IDL и др.), и в виде реализации (функции Windows API, утилиты в составе различных SDK, поддержка и широкое использование данной технологии в операционных системах Windows 95/98/NT, вплоть до использования реестра в качестве регистрационной базы данных COM-сервисов и списков пользователей сети при определении прав доступа к сервисам, а также поддержка COM в других программных продуктах Microsoft). Это обусловило широкую популярность COM как технологии, реализующей объектно-ориентированный подход не на уровне реализации кода, а на уровне сервисов и приложений операционной системы, несмотря на ограниченный спектр поддерживаемых этой технологией платформ (пока это различные версии Windows, хотя информация о предстоящих реализациях для других операционных систем уже начинает появляться). В COM, как в технологии организации распределенных вычислений нет, по существу, ничего революционного. И вызовы удаленных процедур, и создание stub- и proxy-объектов, и регистрация сервисов в специализированных базах данных, и язык IDL как средство описания интерфейсов сервера и сервисов - все это было придумано задолго до возникновения COM (и даже задолго до появления Windows).

Однако, COM согласно спецификациям, позволяет решить множество проблем программирования для Windows, таких как:

- существование различных версий одних и тех же библиотек;
- возможность замены новых версий библиотек старыми при установке тех или иных программных продуктов;
- наличие нескольких реализаций одной и той же спецификации сервиса;
- присутствие нескольких сервисов в одной библиотеке;
- и др.

Это также существенно повлияло на популярность COM. Однако подробное обсуждение этих возможностей выходит за рамки данного курса. Интересующиеся этими аспектами могут более подробно ознакомиться с ними на сайте Microsoft (см., например, Brockschmidt K. What OLE is really about, www.microsoft.com/oledev/olecom/aboutole.html).

Более существенным фактором при рассмотрении имеющихся возможностей организации распределенных вычислений является то, что использование для этой цели COM является одним из самых недорогих решений. Регистрационная база данных (реестр) - это составная часть операционной системы, и, соответственно, не нуждается в отдельном приобретении; поддержка DCOM (Distributed COM) в виде соответствующих сервисов либо также присутствует в операционной системе (Windows NT), либо доступна бесплатно (Windows 95). Сервисы, занимающиеся поиском одной из нескольких реализаций сервера для данного клиента (directory services), в DCOM как таковом отсутствуют - местоположение реализации сервера фиксируется при настройке DCOM для конкретного клиента (есть, конечно, надстройки над COM, обеспечивающие такой сервис, например, Inprise OLEnterprise, но их использование не является обязательным).

Из этого, конечно, не следует, что распределенная информационная система с помощью COM/DCOM может быть создана бесплатно. Если удаленный сервер предоставляет клиентам сервисы доступа к данным, приобретению подлежат лицензии на клиентскую часть серверной СУБД (при этом их число может быть равным не числу серверов, а числу конечных пользователей - все определяется лицензионным соглашением производителя серверной СУБД). Помимо этого, могут быть и другие лицензии, подлежащие приобретению в этом случае, например, лицензия на многопользовательский доступ к Borland Database Engine, входящая в состав продукта Inprise MIDAS. Однако даже с учетом этих затрат общая стоимость такой информационной системы оказывается существенно ниже, чем при использовании, например, Inprise Entera. Естественно, чрезвычайно высоких требований к надежности систем на основе COM при этом предъявлять не стоит, но во многих случаях такое решение может оказаться вполне удовлетворительным.

В настоящее время популярным направлением развития информационных систем для малых и средних предприятий является создание трехзвенных систем с использованием технологии Inprise MIDAS, базировавшейся до недавнего времени на том, что серверы доступа к данным представляют собой не что иное, как COM-серверы. Они называются также серверами автоматизации и поддерживают интерфейс IDataBroker (сейчас серверы MIDAS могут быть не только COM-, но и CORBA-серверами). В данном курсе не содержится детальных подробностей создания обычных MIDAS-серверов и клиентов. Однако, при создании многозвенных систем с помощью этой технологии в промышленных масштабах могут возникнуть некоторые проблемы.

Проблемы эксплуатации COM-серверов и COM+

Разработчики COM-серверов нередко сталкиваются с различными проблемами при их создании и эксплуатации. В частности, при разработке COM-серверов для доступа к данным, обслуживающих нескольких клиентов, следует позаботиться о поддержке нескольких соединений с базой данных и о работе с несколькими потоками. Создание подобного кода с помощью удаленных модулей данных Delphi или C++Builder, содержащих компоненты TDatabase и TSession, не представляет особых сложностей. Однако при большом числе обслуживаемых клиентов наличие подобного многопользовательского сервиса предъявляет серьезные требования к аппаратному обеспечению компьютера, на котором этот сервис функционирует. Поэтому нередко разработчики пытаются создать дополнительный код для осуществления совместного доступа многих клиентов к нескольким соединениям с базой данных, при этом число последних должно быть по возможности минимальным. Обычно для такого разделения ресурсов используется термин "database connection pooling", и в комплекте поставки Delphi 4 Client/Server Suite имеется соответствующий пример.

При подключении очередного клиента к COM-серверу происходит создание обслуживающего его COM-объекта (например, удаленного модуля данных), и этот объект при отключении клиента от сервера уничтожается. В известном смысле такой объект

является "личным" объектом данного клиента. Заметим, что создание серверных объектов по запросу клиента требует ресурсов (оперативной памяти, времени), что становится актуальным в условиях реальной промышленной эксплуатации многозвенных систем, когда удаленные модули данных или иные подобные объекты обслуживают большие объемы данных из большого количества таблиц. Поэтому для экономии времени, затрачиваемого на создание и уничтожение таких COM-объектов, имеет смысл создать дополнительный код, осуществляющий однократное создание нескольких подобных COM-объектов коллективного пользования и предоставляющий их на время обратившимся клиентам по их запросу.

Еще одна проблема, с которой сталкиваются разработчики приложений, предназначенных для работы с серверными СУБД - обработка транзакций, представляющих собой изменение данных в нескольких таблицах, которые либо все вместе выполняются, либо все вместе отменяются. Нередко код, описывающий транзакцию в стандартной двухзвенной клиент серверной системе, содержится в клиентском приложении, а не в серверной части, просто потому, что в случае отката транзакции клиентское приложение должно быть уведомлено об этом. Что касается распределенных транзакций, использующих синхронные изменения в нескольких разных базах данных, их обработка практически всегда производится только в клиентском приложении. Подобные требования усложняют написание клиентских приложений, особенно если в информационной системе их несколько, и повышают соответствующие требования к аппаратной части рабочих станций. Нередко с целью изъятия кода обработки транзакций из клиентского приложения разработчики создают специализированные сервисы, ответственные за обработку транзакций (так называемые мониторы транзакций; есть и специальные продукты, предназначенные для управления распределенными транзакциями).

Имеется также ряд проблем, связанных с авторизованным доступом пользователей к сервисам, предоставляемым COM-серверами. Эти вопросы, если рассматривать их в рамках традиционной COM-технологии, остаются исключительно на совести разработчиков этих сервисов, а также системных администраторов, конфигурирующих DCOM. Спецификация COM не содержит никаких требований на этот счет.

Таким образом, возникла потребность в расширении COM-технологии. Расширение заключалось в разработке сервиса, который бы обеспечивал:

- создание COM-объектов для совместного использования многими клиентами;
- поддерживал авторизованный доступ к этим объектам, а также при необходимости обработку транзакций этими объектами.

Расширенная таким образом технология COM получила название **COM+**, а сам сервис, реализующий это расширение, получил название **Microsoft Transaction Server (MTS)**.

Microsoft Transaction Server представляет собой сервис, обеспечивающий централизацию использования серверов автоматизации, а также управление транзакциями и совместное использование несколькими клиентами соединений с базой данных независимо от реализации сервера. Версия 2.0 этого сервиса входит в состав NT Option Pack и доступна для Windows NT и Windows 95/98. Однако некоторые возможности MTS (например, управление удаленными объектами) реализованы только в версии NT Option Pack для Windows NT.

Отметим, что, помимо создания COM-объектов для коллективного пользования, предоставления сервисов авторизации пользователя при доступе к объектам и обработки транзакций, MTS предоставляет средства мониторинга объектов и транзакций, что упрощает их реализацию.

Организация распределенных вычислений с помощью Inprise Entera

Возможности создания распределенных систем ограничиваются не только использованием COM-технологии и, соответственно, MIDAS-ориентированными серверами. Нередко с целью обеспечения повышенной надежности серверы функциональности разрабатывают для платформ, отличных от Windows, и в этом случае используются технологии, отличные от COM.

В данной гоаве будет рассмотрено создание серверов функциональности с помощью Inprise Entera, представляющего собой сервер приложений, функционирующий на многих платформах, таких, как AIX, HP-UX, Solaris, Windows NT. Если говорить более точно, Entera представляет собой набор сервисов и утилит для создания и эксплуатации серверов функциональности и их клиентов, при этом и серверы, и клиенты могут быть созданы на различных языках программирования и для различных платформ. Это позволяет создавать многоуровневые системы в гетерогенной среде, где не только серверы баз данных, но и любые другие серверы функциональности могут выполняться под управлением платформ, отличных от Windows (например, на UNIX-платформах), что позволяет выбрать наиболее оптимальное для предприятия сочетание удобства, масштабируемости и надежности.

Назначение и состав Inprise Entera

Основное назначение серверов функциональности, созданных на базе Entera - предоставлять содержащим только интерфейс пользователя "тонким" клиентским приложениям те или иные услуги, например, проведение сложных расчетов или доступ к данным, содержащимся на серверах баз данных. Будучи сервером доступа к данным (или иным сервером функциональности) для клиентского приложения, серверы на базе Entera могут быть, в свою очередь, клиентами серверных СУБД. Таким образом, с использованием Entera возможно построение трехзвенной системы, где в среднем звене содержатся средства доступа к данным, а также, при необходимости, бизнес-правила, в том числе бизнес-правила, оформленные в виде исполняемых файлов, выполнение которых инициируется при необходимости. При этом нередко среднее звено состоит из комплекса серверов приложений, функционирующих на нескольких компьютерах, нередко под управлением различных операционных систем (рис. 3.1)

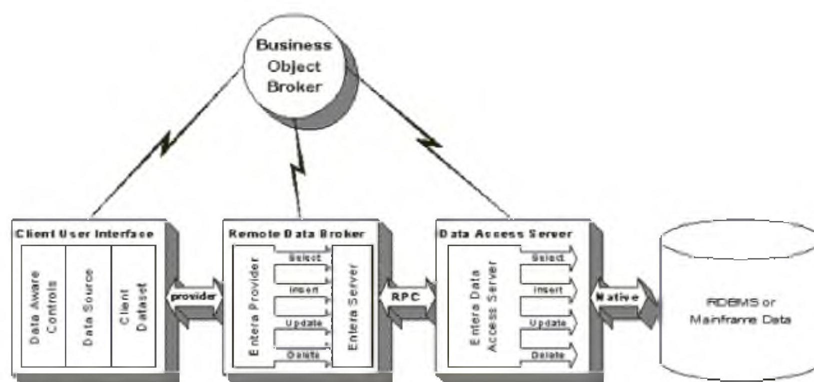


Рис. 3.1. Архитектура многозвенной системы с использованием Entera

Entera поддерживает как стандарт распределенных вычислений DCE (Distributed Computing Environment), так и обмен данными между клиентом и сервером функциональности непосредственно с помощью протокола TCP/IP (только версия 3.2), позволяя при этом создавать клиенпские приложения с помощью Delphi, Visual Basic, PowerBuilder, Smalltalk, Visual C++, Java, COBOL, C, C++, а также средств разработки 4-го поколения (в состав Entera входят соответствующие генераторы клиентского и серверного stub-кода). Поддерживается также широкий спектр серверных СУБД: Oracle, Sybase,

Informix, Ingres, IBM DB2, Microsoft SQL Server. Пользователям Entera доступны соответствующие сервисы доступа к данным, представляющие собой надстройки над клиентскими частями этих серверных СУБД.

Entera включает в себя, помимо генераторов кода и сервисов доступа к данным, специализированные сервисы, обеспечивающих надежность и производительность многозвенных информационных систем.

Entera Broker предоставляет клиентскому приложению список доступных в сети сервисов, оформленных в виде объектов, и находит для обратившегося клиента из этого списка один из них. Entera Broker представляет собой, по существу, так называемый *Directory Service.*

Сервисы безопасности (**Security Services**) обеспечивают доступ пользователей к этим объектам в соответствии с их правами (их обсуждение выходит за рамки данного курса).

AppCenter, или сервис управления приложениями, обеспечивает надежность функционирования клиентских приложений, подключая их в случае отказа компьютеров, содержащих используемые ими объекты, к другим аналогичным объектам, доступным в сети. Этот сервис позволяет также описывать правила функционирования сервисов и групп сервисов (например, равномерная загрузка серверов, или резервирование отказавшего сервера, и др.)

Помимо стандартного комплекта Entera Developers Package, разработчикам доступен ряд дополнительных продуктов: **DCE Adapter** - средство, предоставляющее возможность использования DCE клиентскими приложениями; **Entera/Fx -** набор утилит, включающих дополнительны средства повышения безопасности, средства равномерного распределения загрузки серверов приложений, и др.

Лекция 4. Использование CORBA для организации распределенных вычислений

Данная лекция посвящена организации распределенных вычислений с помощью технологии CORBA (Common Object Request Broker Architecture). Спецификация CORBA разработана консорциумом OMG (Object Management Group).

Так же как и COM, CORBA реализует концепцию объектно-ориентированного подхода и повторного использования кода не на уровне наследования реализации классов внутри одного приложения, а на уровне разных приложений и операционной системы. Однако, в отличие от COM, стандарт CORBA применим не только в случае различных версий Windows, но и в случае других платформ.

Спецификация CORBA определяет правила взаимодействия клиентов с объектами, реализованными в серверах. Для осуществления этого взаимодействия используется Object Request Broker, представляющий собой, в отличие от брокеров OLEnterprise или Entera (конкретных приложений) концепцию, которая может иметь несколько разных реализаций, имеющих, в свою очередь, в своем составе определенный набор сервисов. В случае Delphi обычно используется реализация Visibroker. Более подробно о сервисах CORBA, содержащихся в этой реализации, будет изложено ниже.

Создание CORBA-серверов, переносимых на другие платформы, с помощью C++Builder 4

Данный параграф посвящен возможностям C++Builder 4, связанным с созданием распределенных многоплатформенных систем. При организации распределенных вычислений за последние годы стали популярны распределенные системы на основе COM и MIDAS (возможность создания которых появилась в предыдущей версии C++Builder). Однако нередко возникает потребность использовать в качестве серверной части такой распределенной системы приложения, функционирующие на платформе, отличной от

Windows, а это исключает использование COM как специфичной для Windows технологии.

C++Builder позволяет генерировать код для таких переносимых приложений (естественно, за счет отказа от использования COM, VCL и Windows API). Соответственно в качестве технологии распределенных вычислений в этом случае используется многоплатформенная технология, не использующая специфику Windows. В последнее время наиболее часто для этой цели используется CORBA. В комплект поставки C++Builder Enterprise входит набор утилит и сервисов, предстаавляющих собой одну из реализаций этой спецификации - Inprise VisiBroker.

Данная часть обзора возможностей C++Builder 4 описывает создание сервера функциональности с использованием VisiBroker, переносимого на другие платформы, а также создание клиентских приложений для него, в том числе переносимых на другие платформы.

Постановка задачи

Создадим приложение, производящее, к примеру, сложные научные вычисления. Для примера возьмем вычисление значения некоторой функции, представляющей собой полином третьей степени. В реальной жизни вместо этого расчета данная функция может реализовывать любую другую функциональность - доступ к данным, обработку сигналов, поступающих с аппаратуры, генерацию отчетов, и др. Пусть наше приложение имеет главную форму следующего вида (рис. 4.1):

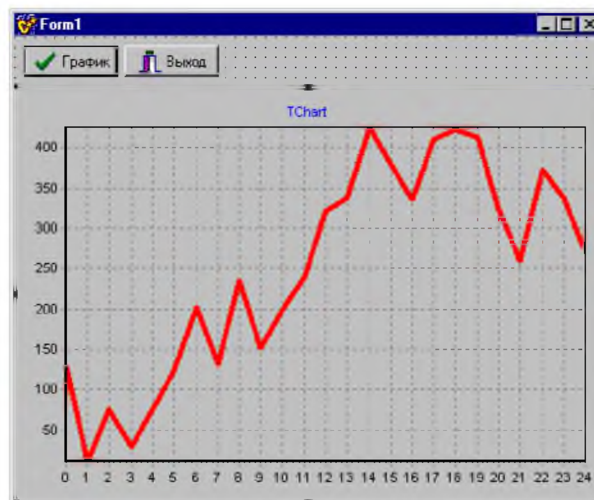


Рис. 4.1. Интерфейс приложения, подлежащего разбиению на сервер и клиента

В нашем примере компонент TChart содержит одну серию (ее следует добавить вручную при создании формы).

Создадим обработчик события, связанный с нажатием на кнопку с надписью "График", и в этом же модуле создадим реализацию нашей функции:

```
//-----------------------------------------------------------------
-
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//-----------------------------------------------------------------
-
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
```

```
//---------------------------------------------------------------------
-
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
-
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  int i; double x1,y;
 for (i=1;i<60;i++)
 {
  x1=0.1*float(i-13);
  y=fun1(x1);
  Chart1->Series[0]->AddXY(x1,y,FloatToStr(x1),clWhite);
 }
}
 double fun1(double x)
{
 double r=x*x*x-5*x*x+3*x+5;
 return(r);
}
//---------------------------------------------------------------------
-
```

Результатом работы такого приложения будет график нашего полинома, появляющийся при нажатии на кнопку "График" (рис. 4.2):

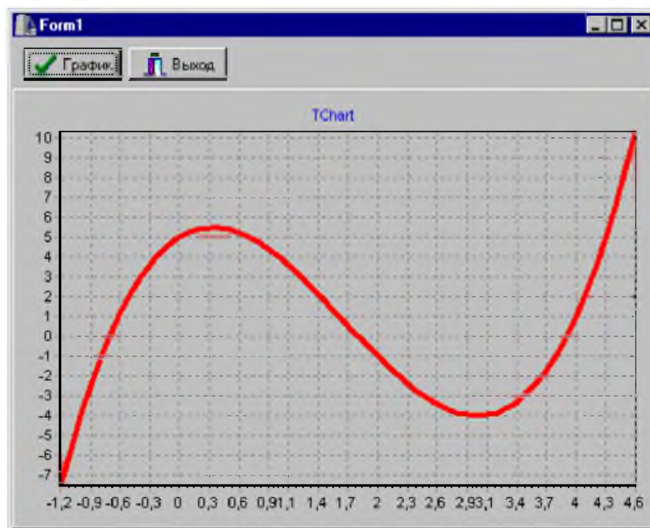


Рис. 4.2. То же приложение на этапе выполнения

Это обычное Windows-приложение, содержащее в себе и пользовательский интерфейс, и функциональность, связанную с проведением расчетов (вычисление значений функции).

Теперь попробуем разбить это приложение на две части, отделив пользовательский интерфейс (кнопки и компонент TChart) от функциональности. Иными словами, создадим сервер функциональности, вычисляющий значение функции, и клиентское приложение, использующее этот сервер как поставщика результатов расчетов.

Создание переносимого сервера функциональности

Создадим CORBA-сервер, осуществляющий проведение расчетов. Сразу же заметим, что подобного рода серверы, как правило, не нуждаются в пользовательском

интерфейсе. Поэтому мы вполне можем создать его как консольное приложение. Если при этом мы откажемся от использования VCL, код полученного сервера может быть скомпилирован любым другим компилятором C++ (в том числе, естественно, и компилятором для другой платформы). Консольные приложения есть единственный тип приложений, существующий для любых платформ, в отличие от приложений с графическим интерфейсом пользователя - последние требуют графических библиотек или операционных систем с GUI, и на все эти графические библиотеки и функции API подобных операционных систем нет никаких стандартов, в отличие от самого языка C++. Иными словами, сервер может быть создан переносимым на другие платформы, что и будет сделано.

Для создания сервера выберем из репозитария объектов со страницы Multitier пиктограмму CORBA Server (рис. 4.3):

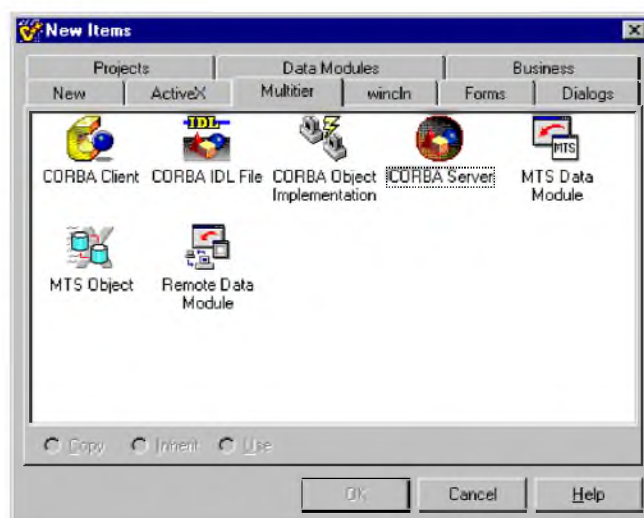


Рис. 4.3. Выбор пиктограммы CORBA Server из репозитария объектов

После выбора пиктограммы в диалоговой панели CORBA Server Wizard выберем в качестве типа приложения Console Application и отключим возможность использования VCL. Это позволит нам создать переносимый на другие платформы код (рис. 4.4):

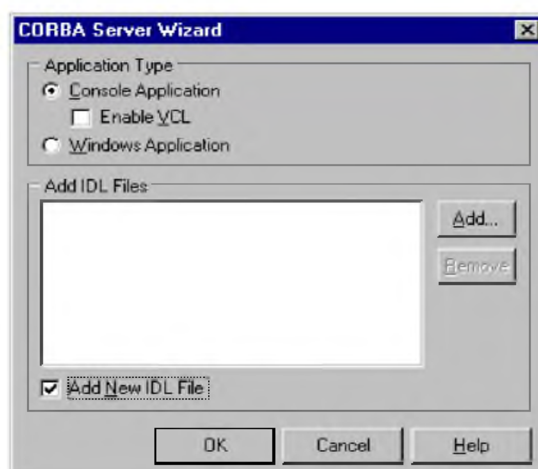


Рис. 4.4. CORBA Server Wizard

Первое, с чего следует начинать создание распределенной системы, - это описание интерфейсов сервера. С этой целью используется язык IDL (Interface Definition Language), являющийся, по существу, стандартом для подобного рода описаний, не зависящим от языков программирования и платформ. Отметим, что существует несколько диалектов IDL (COM IDL, CORBA IDL, DCE IDL), имеющих некоторые различия. Естественно, мы

будем использовать CORBA IDL. Так как мы еще не создали никакого IDL-описания, выберем опцию Add New IDL File. После нажатия на кнопку OK в окне редактора кода появится пустая страница для ILD-описания, куда можно ввести описание интерфейса нашего сервера.

```
interface b1
{
 double fun1(in double x);
 } ;
```

Далее следует скомпилировать IDL-файл. В действительности это не компиляция, а кодогенерация, в результате которой мы получим два модуля, fun1_s.cpp and fun1_c.cpp, содержащие stub-код и skeleton-код.

Stub-объект и skeleton-объект

Все способы взаимодействия между серверами функциональности и их клиентами основаны на механизме вызовов удаленных процедур (RPC - Remote Procedure Calls) и маршалинга, представляющего собой обмен пакетами данных между объектом внутри клиента (stub-объектом) и объектом внутри сервера (skeleton-объектом). Skeleton-объект является представителем клиента внутри адресного пространства сервера. Обращаясь к нему, сервер "думает", что он имеет дело с локальным объектом. Stub-объект является представителем сервера внутри адресного пространства клиентского приложения, поэтому последнее также "думает", что вызывает методы локального объекта. В действительности так оно и есть, однако вместо истинной реализации (в нашем случае - вычисления значений функции) методы этого объекта осуществляют вызовы удаленных процедур и обмен пакетами данных с сервером (рис. 4.5):

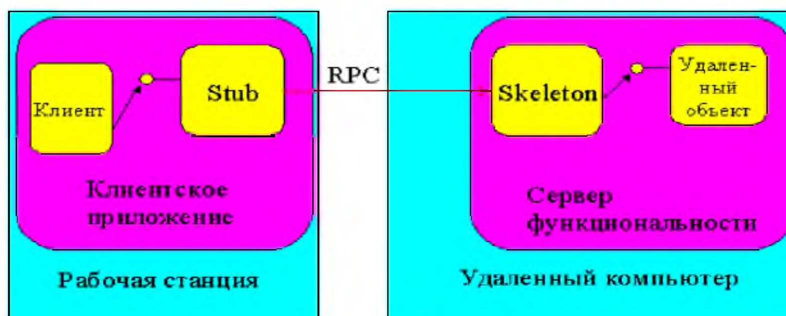


Рис. 4.5. Взаимодействие сервера и клиента

Теперь можно вспомнить о реализации нашей функции. Для этого выберем со страницы Multitier репозитария объектов пиктограмму CORBA Object Implementation. В диалоговой панели CORBA Object Implementation Wizard выберем имя интерфейса (b1), а также определим имя модуля, содержащего реализацию, и имя класса CORBA-объектов. В данном примере опишем объект, экземпляр которого создается в момент старта сервера, так что сервер сразу же сможет принимать обращения клиентов (рис. 4.6).

Можно просмотреть изменения, внесенные в наше приложение этим экспертом. В частности, имеет смысл внести в код реализации нашего метода строки, ответственные за проведение расчетов (рис. 4.7).

Теперь можно скомпилировать проект и закрыть его. Перед созданием клиента мы должны запустить VisiBroker Smart Agent - службу каталогов CORBA, позволяющую осуществить доступ к CORBA-серверам. Теперь можно запустить наш сервер (желательно отдельно от среды разработки).

После этого можно создавать клиентские приложения.

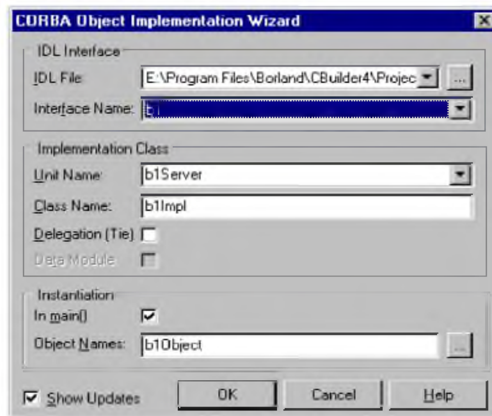


Рис. 4.6. CORBA Object Implementation Wizard

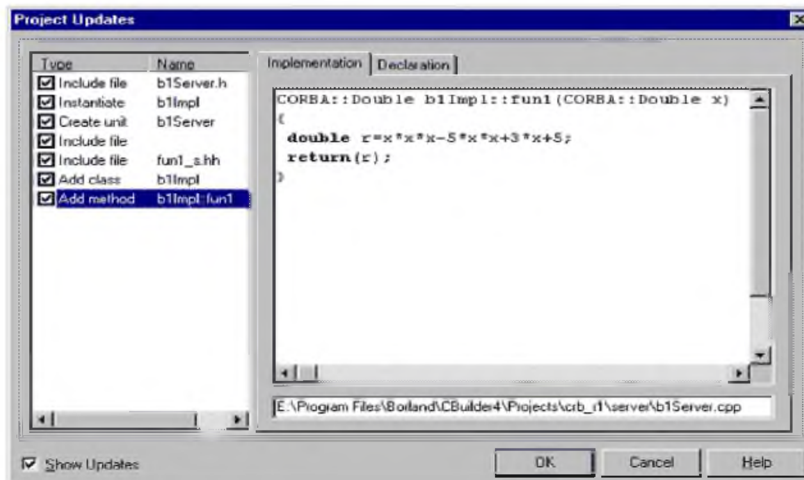


Рис. 4.7. Добавление кода реализации функции в сгенерированный модуль

Создание клиента с графическим пользовательским интерфейсом

Для начала создадим клиентское приложение, представляющее собой обычное Windows-приложение с интерфейсом, похожим на представленный на рис.4.2. Для этого выберем пиктограмму CORBA Client со страницы Multitier репозитария объектов C++Builder.

В этом примере будем использовать раннее связывание клиента с сервером (в этом случае можно достичь максимального быстродействия). Для этого выберем из меню среды разработки опцию "Edit/Use CORBA object". В соответствующем эксперте мы должны добавить к проекту тот же самый IDL-файл, что был создан при создании сервера, а также ответить на вопросы об именах объектов и переменных, в частности, об имени нового свойства формы, к которому следует обращаться, если нужно вызвать метод сервера (рис. 4.8):

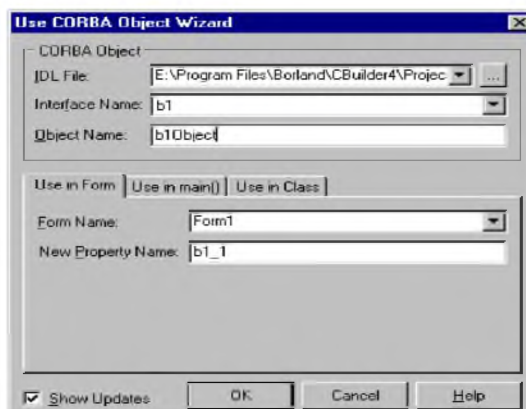


Рис. 4.8. Определение свойств для доступа к CORBA-объекту в клиентском приложении

Далее можно создать интерфейс, сходный с приведенным на рис. 1, и создать обработчик события, связанного с нажатием на кнопку "График":

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
 int i; double x1,y;
 for (i=1;i<60;i++)
 {
  x1=0.1*float(i-13);
  y=b1_1->fun1(x1);
  Chart1->Series[0]->AddXY(x1,y,FloatToStr(x1),clWhite);
         }
}
```

В результате получим тот же график, что был рассмотрен выше.

Создание переносимого клиентского приложения

Теперь создадим клиентское приложение, переносимое на другие платформы. Это должно быть консольное приложение, не использующее VCL. В этом случае при создании клиента мы должны выбрать опцию "Console Application" и отменить опцию "Enable VCL" .

Все остальные действия похожи на предыдущие, за исключением создания пользовательского интерфейса. Простейший способ создания пользовательского интерфейса в данном случае - вывести результаты расчетов на экран и поместить этот код непостредственно в файл проекта:

```
//------------------------------------------------------------------
-
#include <corbapch.h>
#pragma hdrstop
//------------------------------------------------------------------
-
#include "fun1_c.hh"
#include <corba.h>
#include <condefs.h>
USEIDL("corba\corba_rus\fun1.idl");
USEUNIT("corba\corba_rus\fun1_c.cpp");
USEUNIT("corba\corba_rus\fun1_s.cpp");
//------------------------------------------------------------------
-
#pragma argsused
int main(int argc, char* argv[])
{
        try
        {
                 // Initialize the ORB and BOA
                 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
                 CORBA::BOA_var boa = orb->BOA_init(argc, argv);
                 a1_var a1_1 = a1::_bind("a1Obj");

cout<<"Our function table \n";
int i; double x1,y;
for (i=1;i<271;i++)
{
 x1=0.1*float(i);
  y=a1_1->fun1(x1);
  cout<<x1<<" "<<y<<"\n";
   }
```

```
        }
        catch(const CORBA::Exception& e)
        {
                cerr << e << endl;
                return(1);
        }
        return 0;
}
//------------------------------------------------------------------
-
```

В этом случае мы получим таблицу с результатами расчетов непосредственно на экране консольного приложения. Отметим, что данное приложение можно скомпилировать любым компилятором C++, в том числе и компилятором для платформы, отличной от Windows.

Серверы доступа к данным

Серверы доступа к данным являются одним из наиболее популярных типов серверов middleware. Наиболее популярный способ создания переносимого сервера доступа к данным заключается в использовании MIDAS и создание удаленных модулей данных как COM- или CORBA-объектов. В этом случае создается COM или CORBA-сервер, но это будет Windows-приложение, так как оно использует VCL, основанную на Windows API, и BDE, которые также представляет собой набор Windows-библиотек. Переносимый сервер доступа к данным не должен использовать ни VCL, ни BDE. В этом случае, можно использовать то, что функциональность сервера доступа к данным - это набор SQL-запросов, и ничего более. Для MIDAS-серверов генератором запросов является BDE. Но выполнить запрос можно и другим способом. Например, всегда есть возможность использования низкоуровневого API клиентских частей серверных СУБД (например, Oracle Call Interface). Для большинства таких СУБД такие API существуют для многих платформ и содержат одни и те же функции. Использование таких API может показаться несколько утомительным, но в целом оно ненамного сложнее, чем написание других функций.