Министерство образования и науки, молодежи и спорта Украины
Харьковский национальный университет

Широкопетлева М.С., Черепанова Ю.Ю.

ПИ

Распределенные базы данных

ПИ

Харьков
2012

Содержание

Теория

Распределенное хранение данных

Под ***распределенной*** (Distributed DataBase - DDB) обычно подразумевают базу данных, включающую фрагменты из нескольких баз данных, которые располагаются на различных узлах сети компьютеров, и, возможно управляются различными СУБД. Распределенная база данных выглядит с точки зрения пользователей и прикладных программ как обычная локальная база данных. В этом смысле слово "распределенная" отражает способ организации базы данных, но не внешнюю ее характеристику.

РБД состоит из набора узлов, связанных коммуникационной сетью, в которой:

- каждый узел - это полноценная СУБД сама по себе;

- узлы взаимодействуют между собой таким образом, что пользователь любого из них может получить доступ к любым данным в сети так, как будто они находятся на его собственном узле.

Каждый узел сам по себе является системой базы данных. Любой пользователь может выполнить операции над данными на своём локальном узле точно так же, как если бы этот узел вовсе не входил в распределённую систему. Распределённую систему баз данных можно рассматривать как партнёрство между отдельными локальными СУБД на отдельных локальных узлах.

Фундаментальный принцип создания распределённых баз данных («правило 0»): Для пользователя распределённая система должна выглядеть так же, как нераспределённая система.

Фундаментальный принцип имеет следствием определённые дополнительные правила или цели. Перечислим цели (по Дейту):

-Локальная независимость. Узлы в распределённой системе должны быть независимы, или автономны. Локальная независимость означает, что все операции на узле контролируются этим узлом.

-Отсутствие опоры на центральный узел. Локальная независимость предполагает, что все узлы в распределённой системе должны рассматриваться как равные. Поэтому не должно быть никаких обращений к «центральному» или «главному» узлу с целью получения некоторого централизованного сервиса.

-Непрерывное функционирование. Распределённые системы должны предоставлять более высокую степень надёжности и доступности.

-Независимость от расположения. Пользователи не должны знать, где именно данные

хранятся физически и должны поступать так, как если бы все данные хранились на их собственном локальном узле.

-Независимость от фрагментации. Система поддерживает независимость от фрагментации, если данная переменная-отношение может быть разделена на части или фрагменты при организации её физического хранения. В этом случае данные могут храниться в том месте, где они чаще всего используются, что позволяет достичь локализации большинства операций и уменьшения сетевого трафика.

-Независимость от репликации. Система поддерживает репликацию данных, если данная хранимая переменная-отношение - или в общем случае данный фрагмент данной хранимой переменной-отношения - может быть представлена несколькими отдельными копиями или репликами, которые хранятся на нескольких отдельных узлах.

-Обработка распределённых запросов. Суть в том, что для запроса может потребоваться обращение к нескольким узлам. В такой системе может быть много возможных способов пересылки данных, позволяющих выполнить рассматриваемый запрос.

-Управление распределёнными транзакциями. Существует 2 главных аспекта управления транзакциями: управление восстановлением и управление параллельностью обработки. Что касается управления восстановлением, то чтобы обеспечить атомарность транзакции в распределённой среде, система должна гарантировать, что все множество относящихся к данной транзакции агентов (агент - процесс, который выполняется для данной транзакции на отдельном узле) или зафиксировало свои результаты, или выполнило откат. Что касается управления параллельностью, то оно в большинстве распределённых систем базируется на механизме блокирования, точно так, как и в нераспределённых системах.

-Аппаратная независимость. Желательно иметь возможность запускать одну и ту же СУБД на различных аппаратных платформах и, более того, добиться, чтобы различные машины участвовали в работе распределённой системы как равноправные партнёры.

-Независимость от операционной системы. Возможность функционирования СУБД под различными операционными системами.

-Независимость от сети. Возможность поддерживать много принципиально различных узлов, отличающихся оборудованием и операционными системами, а также ряд типов различных коммуникационных сетей.

-Независимость от типа СУБД. Необходимо, чтобы экземпляры СУБД на различных узлах все вместе поддерживали один и тот же интерфейс, и совсем необязательно, чтобы это были копии одной и той же версии СУБД.

Типы распределённых баз данных:

-Распределённые базы данных.

-Мультибазы данных с глобальной схемой. Система мультибаз данных - это распределённая

система, которая служит внешним интерфейсом для доступа ко множеству локальных СУБД или структурируется, как глобальный уровень над локальными СУБД.

-Федеративные базы данных. В отличие от мультибаз не располагают глобальной схемой, к которой обращаются все приложения. Вместо этого поддерживается локальная схема импорта-экспорта данных. На каждом узле поддерживается частичная глобальная схема, описывающая информацию тех удалённых источников, данные с которых необходимы для функционирования.

-Мультибазы с общим языком доступа - распределённые среды управления с технологией «клиент-сервер».

Логическая модель РБД

Логическая модель РБД строится на 3-х уровнях (слоях) абстракцииданных: представления информации, обработки (бизнес-логики) ихранения. Слои образуют строгую иерархию: слой бизнес -логики взаимодействует со слоями хранения и представления. Физически, слои могут входить в состав одного программного модуля, или же распределяться на нескольких параллельных процессах в одном или нескольких узлах сети.

Слой представления информации

Обеспечивает интерфейс с пользователем. Как правило, получение информации от пользователя происходит посредством различных форм. А выдача результатов запросов - посредством отчетов.

Слой бизнес-логики

Связующий, именно он определяет функциональность и работоспособность системы в целом. Блоки программного кода распределены по сети и могут использоваться многократно (CORBA, DCOM) для создания сложных распределенных приложений.

Слой хранения данных

Обеспечивает физическоехранение, добавление, модификацию и выборку данных. На данный слой также возлагается проверка целостности и непротиворечивости данных, а также реализацию разделенных транзакций.

Слои распределенной системы могут быть по разному реализованы и исполняться в разных узлах сети. Обычно рассматриваются следующие архитектуры (табл.1)

Таблица 1 - Архитектуры слоев

Слой \ Тип архитектуры	Файл-сервер	Клиент-сервер (Бизнес-логика	Клиент-сервер (бизнес-логика на	N-уровневая архитектура

		на клиенте)	сервере)	
Представления	Клиент	Клиент	Клиент	Клиент
Бизнес- логики	Клиент	Клиент	Сервер БД	Сервер приложений (комп. кластер)
Хранения	Файл-сервер (или клиент)Все три слоя образуют единый программный модуль	Сервер БДПользоват. Интерфейс и бизнес-логика образуют единый модуль. Данные хранятся на сервере БД	Сервер БДВся бизнес логика реализована в виде хранимых процедур, исполняемых на сервере БД	Сервер БДВсе слои исполняются на разных машинах.

Файл-сервер

В системах, построенных по архитектуре файл-сервера все слои системы представляют единое и неделимое целое. БД хранится в виде файла или набора файлов на файл-сервере. Вся логика выборки, хранения и обеспечения непротиворечивости данных возлагается на клиентскую часть. Файл-серверные системы ориентированы на работу с отдельными записями в таблице.

Достоинтва:

- простота логики;

- низкие требования к аппаратному обеспечению и малый объем требуемой памяти.;

- не требуют надежных многозадачных и многопользовательских ОС.;

- невысокая цена СУБД.

Недостатки:

- ограниченность языка и негибкость среды разработки приложений;

- слабая масштабируемость;

- не обеспечивают многопользовательский режим работы;

- трудно поддерживать целостность и непротиворечивость данных;

- необходимость ручной блокировки записей или таблиц целиком;

- низкий уровень защищенности как внешней (от взлома), так и внутренней (ошибок приложений), например, индексы отдельно от таблиц;

- не имеют средств шифрации сетевого трафика;

- создают высокую нагрузку на сеть.

Клиент-сервер с бизнес-логикой на клиенте

В данных системах хранение, выборка и поддержание непротиворечивости данных возлагается на сервер БД, а вся бизнес-логика и логика представления исполняются на клиентских машинах. Так как все операции по манипулированию данными осуществляются только через сервер, производительность и сохранность данных зависит только от сервера БД. Серверы БД изначально рассчитаны на многопользовательский режим работы, имеют эффективные алгоритмы кеширования данных. Современные серверы имеют хорошую масштабируемость.

Клиентская часть обменивается данными с сервером посредством SQL запросов. Обработка информации в клиент-серверных системах ведется на уровне множества кортежей.

Процесс разработки разделяется на создание БД и написание клиентской части с бизнес-логикой.

Достоинства:

- высокая производительность, стабильность и надежность при многопользовательской работе;

- легко организуется защита данных (шифрование сетевого трафика SSH, SSL);

- универсальность языка определения и манипулирования данными.

Недостатки:

- более высокая цена СУБД. (сервер БД продается отдельно);

- достаточно высокие требования к квалификации разработчиков;

- навыки администрирования сервера БД;

- повышенные требования к пропускной способности сети;

- повышенные требования к клиентским местам (на них выполняется слой бизнес- логики).

Итак, при количестве пользователей от 2 до ~50 она является хорошим вариантом. С ростом числа пользователей начинает сказываться недостаточная пропускная способность сети.

Клиент-сервер с бизнес-логикой на сервере

Используется возможность современных серверов БД исполнять хранимые SQL процедуры

на сервере, куда и переносится максимально возможная часть бизнес-логики. Требования к серверу БД возрастают, однако резко понижаются требования к клиентским машинам (за счет выноса с них бизнес-логики) и к пропускной способности сети (клиенту передаются только данные, необходимые пользователю).

Достоинства:

- пониженные, по сравнению с предыдущим классом систем, требования к пропускной способности сети и клиентским местам;

- более простой процесс создания бизнес-логики.

Недостатки:

- повышенные требования к серверу БД.(каждый сеанс "съедает" память из расчета предельной загрузки);

- невысокая переносимость (мобильность) системы на другие серверы БД.

Следовательно, по сравнению с предыдущими классами, позволяет держать большую нагрузку.

N-уровневая архитектура

Основными элементами являются сервера БД, сервер(кластер) приложений и клиентская часть. Главная идея n-уровневой архитектуры заключается в максимальном упрощении клиента (тонкий клиент) , выносе всей бизнес-логики с клиента и сервера БД.

Тонкий клиент представляет собой некоторый терминал типа HTML-browser или эмуляторы X-терминала

Вся бизнес- логика оформляется в виде набора приложений, запускаемых на сервере приложений под управлением ОС типа UNIX.

Сервера БД занимаются только проблемами хранения, добавления, модификации и поддержания непротиворечивости данных.

Сервер приложений соединен с сервером БД при помощи отдельного высокоскоростного сегмента сети.

Достоинства:

-повышенная защищенность;

-высокая производительность;

-легкость развития и модификации;

-легкость администрирования;

-возможность создания системы с массовым параллелизмом (серверов БД может быть несколько, а сервером приложений могут служить несколько соединенных в кластер компьютеров).

Недостатки:

-высокая сложность;

-высокая цена решения;

-в некоторых случаях уступает по производительности клиент-серверным системам с бизнес-логикой на сервере.

Итак, этот вариант можно рассматривать как единственную альтернативу для создания ИС для очень большого количества пользователей

Архитектура "клиент-сервер"

Распределенные системы - это системы "клиент-сервер". Существует по меньшей мере три модели "клиент-сервер":

-модель доступа к удаленным данным (RDA-модель);

-модель сервера базы данных (DBS-модель);

-модель сервера приложений (AS-модель).

Первые две являются двухзвенными и не могут рассматриваться в качестве базовой модели распределенной системы (ниже будет показано, почему это так). Трехзвенная модель хороша тем, что в ней интерфейс с пользователем полностью независим от компонента обработки данных. Собственно, трехзвенной ее можно считать постольку, поскольку явно выделены:

- компонент интерфейса с пользователем;

- компонент управления данными (и базами данных в том числе),

а между ними расположено программное обеспечение промежуточного слоя (middleware), выполняющее функции управления транзакциями и коммуникациями, транспортировки запросов, управления именами и множество других. Middleware - это **главный** компонент распределенных систем и, в частности, DDB-систем. Главная ошибка, которую мы совершаем на нынешнем этапе - полное игнорирование middleware и использование двухзвенных моделей "клиент-сервер" для реализации распределенных систем.

Существует фундаментальное различие между технологией "SQL-клиент - SQL-сервер" и технологией продуктов класса middleware (например, менеджера распределенных транзакций Tuxedo System). В первом случае клиент явным образом запрашивает данные, зная структуру базы данных (имеет место так называемый data shipping, то есть "поставка данных" клиенту). Клиент передает СУБД SQL-запрос, в ответ получает данные. Имеет место жесткая связь типа "точка- точка", для реализации которой все СУБД используют закрытый SQL-канал (например, Oracle SQL*Net). Он строится двумя процессами: SQL/Net на компьютере - клиенте и SQL/Net на компьютере-сервере и порождается по инициативе клиента оператором CONNECT. Канал закрыт в том смысле, что невозможно, например, написать программу, которая будет шифровать SQL- запросы по специальному алгоритму.

В случае трехзвенной схемы клиент явно запрашивает один из сервисов (предоставляемых прикладным компонентом), передавая ему некоторое сообщение (например) и получает ответ также в виде сообщения. Клиент направляет запрос в информационную шину (которую строит Tuxedo System), ничего не зная о месте расположения сервиса. Имеет место так называемый function shipping (то есть "поставка функций" клиенту). Важно, что для Клиента база данных (в том числе и DDB) закрыта слоем Сервисов. Более того, он вообще ничего не знает о ее существовании, так как все операции над базой данных выполняются внутри сервисов.

Сравним два подхода. В первом случае мы имеем жесткую схему связи "точка-точка" с передачей открытых SQL-запросов и данных, исключающую возможность модификации и работающую только в синхронном режиме "запрос-ответ". Во втором случае определен гибкий механизм передачи сообщений между клиентами и серверами, позволяющий организовывать взаимодействие между ними многочисленными способами.

Таким образом, речь идет о двух принципиально разных подходах к построению информационных систем "клиент-сервер". Первый из них устарел и явно уходит в прошлое. Дело в том, что SQL (ставший фактическим стандартом общения с реляционными СУБД) был задуман и реализован как декларативный язык запросов, но отнюдь не как средство взаимодействия "клиент-сервер" (об этой технологии тогда речи не было). Только потом он был "притянут за уши" разработчиками СУБД в качестве такого средства. На волне успеха реляционных СУБД в последние годы появилось множество систем быстрой разработки приложений для реляционных баз данных (VisualBasic, PowerBuilder, SQL Windows, JAM и т.д.). Все они опирались на принцип генерации кода приложения на основе связывания элементов интерфейса с пользователем (форм, меню и т.д.) с таблицами баз данных. И если для быстрого создания несложных приложений с небольшим числом пользователей этот метод подходит как нельзя лучше, то для создания корпоративных распределенных информационных систем он абсолютно непригоден.

Для этих задач необходимо применение существенно более гибких систем класса middleware (Tuxedo System, Teknekron), которые и составляют предмет нашей профессиональной деятельности и базовый инструментарий при реализации больших проектов.

Проблемы распределенных БД

Исходя из определения Дэйта, распределенную базу данных в общем случае можно

рассматривать как слабосвязанную сетевую структуру, узлы которой представляют собой локальные базы данных. Локальные базы данных автономны, независимы и самоопределены; доступ к ним обеспечивается от различных поставщиков. Связи между узлами - это потоки тиражируемых данных. Топология DDB варьируется в широком диапазоне: возможны варианты иерархии, структур типа звезда и т. д. В целом топология DDB определяется географией информационной системы и направленностью потоков тиражирования данных.

Рассмотрим теперь проблемы реальных распределенных баз данных. Проблемы централизованных СУБД существуют и здесь, однако децентрализация добавляет новые:

В распределенных ИС необходима ***общая логическая модель данных***. Это обеспечит логическую прозрачность данных для пользователя, когда он формирует запрос к БД, не задумываясь, где находятся данные;

Прозрачность размещения - т.е. необходима полная и однозначная схема размещения данных на сети;

Прозрачность преобразования данных, т.к. БД может быть однородной и неоднородной, как по техническим, так и по программным средствам. Поэтому нужно обеспечить совместимость данных, а это возможно лишь путем преобразования их из одного формата в другой;

Прозрачность словарей и управление словарями, т.к. имеется множество справочников и словарей на узлах сети;

Механизм управление запросами, т.к. запросы могут поступать на данные, находящиеся в разных узлах сети;

Сложный механизм управления одновременной обработкой; в частности синхронизация запросов, т.к. запросы могут поступать одновременно к одним и тем же данным;

Развитая методология распределения, размещения и обработки данных. Четкие системные соглашения для архитектуры и между локальными администраторами сети.

База данных физически распределяется по узлам компьютерной информационной системы при помощи фрагментации и репликации (тиражирования) данных.

Транзакции

Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует (COMMIT) изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример информационной системы с файлами СОТРУДНИКИ и ОТДЕЛЫ, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотрудника является объединение элементарных операций над файлами СОТРУДНИКИ и ОТДЕЛЫ в одну транзакцию. Таким

образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД (если, конечно, такая система заслуживает названия СУБД). Но понятие транзакции гораздо более важно в многопользовательских СУБД.

Например, рассмотрим операции, производимые при переводе денег с одного банковского счета на другой. Проведение такой операции требует выполнения следующих действий:

а) Снять деньги с одного счета:

```
UPDATE Bank SET Money=Money -1000 WHERE Chet=100;
```

б) Добавить эту сумму на другой счет:

```
UPDATE Bank SET Money=Money +1000 WHERE Chet=200;
```

в) Зафиксировать операцию перевода денег в банковском журнале:

```
INSERT Operation VALUES (SYSDATE, 100, 200, 1000, … );
```

Эти действия с базой данных банковских счетов должны быть выполнены либо все полностью, либо ни одно из них не должно быть выполнено. Таким образом, вышеперечисленные 3 операции необходимо рассматривать, как транзакцию.

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

С управлением транзакциями в многопользовательской СУБД связаны важные понятия сериализации транзакций и сериального плана выполнения смеси транзакций. Под сериализаций параллельно выполняющихся транзакций понимается такой порядок планирования их работы, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Сериальный план выполнения смеси транзакций - это такой план, который приводит к сериализации транзакций. Понятно, что если удается добиться действительно сериального выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. В централизованных СУБД наиболее распространены алгоритмы, основанные на синхронизационных захватах объектов БД. При использовании любого алгоритма сериализации возможны ситуации конфликтов между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержания сериализации необходимо выполнить откат (ликвидировать все изменения,

произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

Свойства транзакций

Атомарность - транзакция является неделимой, она выполняется полностью или не выполняется вообще; если транзакция прерывается на середине, то происходит откат (ROLLBACK) и база данных должна остаться в том состоянии, которое она имела до начала транзакции;

Параллельность - эффект от параллельного выполнения нескольких транзакций должен быть таким же, как от их последовательного выполнения;

Целостность - транзакция переводит базу данных из одного непротиворечивого (целостного) состояния в другое;

Долговременность - после того, как транзакция завершена и зафиксирована (COMMIT), результат ее выполнения гарантированно сохраняется в базе данных.

Аномалии доступа к базе данных

Потерянные изменения - lost update

Имеется сходная таблица Example1(см. таблица2).

Таблица2 - Example1

ID Integer	Dat Integer
1	100
2	110
3	120
4	130

Выполняются две транзакции, которые меодифицируют одну и ту же запись. В таблице Table2 (таблица 3)приведен порядок выполнения команд этими транзакциями.

Таблица 3 - Table2

Транзакция T1	Шаг	Транзакция T1
Update Example Set dat=dat+1 Where	1	

Id=1;		
	2	Update Example Set dat=dat+1 Where Id=1;
Commit;	3	
	4	Commit;
Select * from Example	5	

В таблице Example2 (таблица 4)продемонстрированы результаты работы транзакций (пошагово). Как видно, значение, записанное транзакцией T2, «затрет» значение, записанное транзакцией T1 (дважды последовательно происходило увеличение значение первой строки, а в результате зафиксировалось только увеличение на 1).

Таблица 4 - Example2

ID Integer	Dat Integer
1	101
2	110
3	120
4	130

Аномалии доступа к базе данных (грязное чтение - dirty read)

Имеется сходная таблица Example1 (таблица 5).

Таблица 5 - Example1

ID Integer	Dat Integer
1	100
2	110
3	120
4	130

Выполняются две транзакции, которые модифицируют одну и ту же запись. В таблице Table2 (таблица 6)приведен порядок выполнения команд этими транзакциями.

Таблица 6 - Table2

Транзакция T1	Шаг	Транзакция T1
	1	Update Example Set dat=101 Where Id=1;
Update Example Set =(Select dat From example Where Id=1)	2	

Where Id=2;		
Select * from Example;	3	
	4	Rollback;
	5	Select * from Example;

В таблице Example2 (таблица 7) продемонстрированы результаты работы на 3 шаге (когда произошло изменение в соответствии с неверной модификацией данных на шаге 1).

Таблица 7 - Example2

ID Integer	Dat Integer
1	101
2	101
3	120
4	130

В результате, после выполнения всех шагов будет получена следующая таблица 8. При этом, считывание информации на разных шагах дало различные результаты (что недопустимо).

Таблица 8 - Example2

ID Integer	Dat Integer
1	100
2	101
3	120
4	130

Аномалии доступа к базе данных (неповторяющееся чтение - non-repeatable read)

Имеется сходная таблица Example1 (таблица 9).

Таблица 9 - Example1

ID Integer	Dat Integer
1	100
2	110
3	120
4	130

Независимо выполняются две транзакции, одна из которых считывает данные, а другая модифицирует. В таблице Table2 (таблица 10 приведен порядок выполнения команд этими транзакциями.

Таблица 10 - Table2

Транзакция T1	Шаг	Транзакция T1
Select * from Example Where Id=1;	1	
	2	Update Example Set dat=101 Where Id=1;
	3	Commit;
Select * from Example Where Id=1;	4	
Commit;	5	

В таблице Example2 (таблица 11) продемонстрированы результаты работы на шаге 1 (считывание информации 1 транзакцией).

Таблица 11 - Example2

ID Integer	Dat Integer
1	100

В результате выполнения запроса на 4 шаге получим таблицу 12. Как видно, чтение одних и тех же данных в транзакции T1 дают разные результаты.

Таблица 12 - Example2

ID Integer	Dat Integer
1	101

Аномалии доступа к базе данных (фантом - phantom insert)

Имеется сходная таблица Example1(таблица 13).

Таблица 13 - Example1

ID Integer	Dat Integer
1	100
2	110
3	120
4	130

Независимо выполняются две транзакции, одна из которых считывает данные, а другая добавляет. В таблице Table2 (таблица 14) приведен порядок выполнения команд этими транзакциями.

Таблица 14 - Table2

Транзакция T1	Шаг	Транзакция T1
Select * from Example Where Dat>110;	1	
	2	Insert Into example Values (5, 140);
	3	Commit;
Select * from Example Where Dat>110;	4	
Commit;	5	

В таблице Example2 (таблица 15) продемонстрированы результаты работы на шаге 1 (считывание инфорамции 1 транзакцией).

Таблица 15 - Example2

ID Integer	Dat Integer
3	120
4	130

В результате выполнения запроса на 4 шаге получим таблицу 16. Как видно, одинаковые запросы в транзакции T1 выбирают разные множества строк.

Таблица 16 - Example2

ID Integer	Dat Integer
3	120
4	130
5	140

Управление транзакциями

Первое предложение SQL в вашей программе начинает транзакцию. Когда одна транзакция заканчивается, очередное предложение SQL автоматически начинает следующую транзакцию. Таким образом, каждое предложение SQL является частью некоторой транзакции. Предложения COMMIT и ROLLBACK гарантируют, что все изменения в базе данных, осуществленные операциями SQL, либо становятся постоянными, либо отменяются единовременно. Все предложения SQL, выполненные после последней операции COMMIT или ROLLBACK, составляют текущую транзакцию. Предложение SAVEPOINT отмечает и именует текущую точку в обработке транзакции.

Распределенные транзакции включают, по меньшей мере, одно предложение SQL, изменяющее данные в нескольких узлах распределенной базы данных. Если обновление затрагивает лишь один узел, транзакция удаленная, но не распределенная. Если часть

распределенной транзакции сбивается, вы должны выполнить откат всей транзакции или откат к точке сохранения. ORACLE в такой ситуации выдает сообщение об ошибке. Поэтому вы должны включать проверки на ошибки в каждое приложение, выполняющее распределенные транзакции.

Использование COMMIT

Предложение ***COMMIT*** завершает текущую транзакцию и делает постоянными все изменения, осуществленные в течение этой транзакции. До этого момента другие пользователи не могут видеть измененных данных; они видят данные в том состоянии, каким оно было к моменту начала транзакции. Рассмотрим простую транзакцию, которая осуществляет перевод денег с одного банковского счета на другой. Эта транзакция требует двух операций обновления (UPDATE), потому что она должна дебитовать один счет и кредитовать другой. После кредитования второго счета вы выдаете команду COMMIT, делая изменения постоянными. Лишь после этого новое состояние счетов становится видимым другим пользователям.

Пример:

```
BEGIN
...
UPDATE accts SET bal = my_bal - debit WHERE acctno = 7715;
...
UPDATE accts SET bal = my_bal + credit WHERE acctno = 7720;
COMMIT WORK;
END;
```

Необязательное ключевое слово WORK не имеет никакого эффекта, помимо улучшения читабельности. Предложение COMMIT освобождает все блокировки таблиц и строк. Оно также стирает все точки сохранения (обсуждаемые ниже), отмеченные после последней операции COMMIT или ROLLBACK. Когда вы выдаете COMMIT, постоянными становятся изменения во всех базах данных, затронутых распределенной транзакцией. Однако, если во время выполнения COMMIT произойдет сбой сети или машины, состояние распределенной транзакции может оказаться неизвестным или СОМНИТЕЛЬНЫМ (in doubt).

Использование ROLLBACK

Предложение ***ROLLBACK*** противоположно COMMIT. Оно заканчивает текущую транзакцию и отменяет все изменения, осуществленные за время этой транзакции.

Предложение ROLLBACK полезно по двум причинам. Во-первых, если вы сделали ошибку, например, удалили не ту строку из базы данных, вы можете использовать ROLLBACK для восстановления первоначальных данных. Вариант ***ROLLBACK TO*** позволяет вам отменить изменения до промежуточной точки в текущей транзакции, так что вы не обязаны стирать все ваши изменения. Во-вторых, предложение ROLLBACK полезно, когда вы начали транзакцию, которую не в состоянии завершить, например, при возникновении исключения или ошибки в

предложении SQL. В таких случаях ROLLBACK позволяет вам вернуться к стартовой точке, так что вы можете предпринять корректирующие действия и попытаться снова повторить транзакцию.

Рассмотрим следующий пример, в котором вы вставляете информацию о сотруднике в три различных таблицы базы данных. Все три таблицы имеют столбец, содержащий номер сотрудника и ограничиваемый уникальным индексом. Если предложение INSERT пытается вставить повторяющийся номер сотрудника, возбуждается предопределенное исключение DUP_VAL_ON_INDEX. В этом случае вам необходимо отменить все изменения. Поэтому вы выдаете ROLLBACK в обработчике исключений (исключения будут рассмотрены в теме "Обработка исключительных ситуаций").

Пример:

```
DECLARE
emp_id INTEGER;
...
BEGIN
SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ...
...
INSERT INTO tab1 VALUES (emp_id, ...);
INSERT INTO tab1 VALUES (emp_id, ...);
INSERT INTO tab1 VALUES (emp_id, ...);
... EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
ROLLBACK ;
...
END ;
```

Когда вы выдаете ROLLBACK, отменяются изменения во всех базах данных, затронутых распределенной транзакцией.

Откаты на уровне предложений

Прежде чем исполнять предложение SQL, ORACLE выдает неявную точку сохранения. Затем, если это предложение сбивается, ORACLE автоматически выполняет его откат. Например, если предложение INSERT пытается вставить повторяющееся значение в уникальный индекс, оно откатывается. При этом теряется лишь работа, начатая сбившимся предложением SQL; вся работа, проделанная в текущей транзакции до этого момента, не затрагивается. ORACLE может также предпринимать откат одиночных предложений SQL с целью предотвратить мертвые захваты (взаимоблокировки). В таких случаях ORACLE сигнализирует об ошибке одной из транзакций и выполняет откат текущего предложения SQL в этой транзакции. Прежде чем исполнять предложение SQL, ORACLE должен выполнить его РАЗБОР (parse), т.е. исследовать его, чтобы убедиться, что оно синтаксически корректно и ссылается на действительные объекты базы данных. Ошибки, обнаруженные во время разбора предложения (в отличие ошибок во

время выполнения) не приводят к откату.

Использование SAVEPOINT

SAVEPOINT отмечает и именует текущую точку (точку сохранения) в процессе транзакции. Такая точка, используемая в предложении ROLLBACK TO, позволяет отменить часть транзакции. В следующем примере вы отмечаете точку сохранения перед тем, как выполнять вставку строки. Если предложение INSERT попытается вставить повторяющееся значение в столбец ware _ id , возникнет предопределенное исключение DUP_VAL_ON_INDEX. В этом случае вы откатитесь к точке сохранения, отменив лишь вставку.

Пример:

```
DECLARE
ware_id1 warehouses.ware_id%TYPE;

BEGIN
...
UPDATE warehouses SET ... WHERE ware_id = ware_id1;
DELETE FROM warehouses WHERE ...
...
SAVEPOINT do_ins;
INSERT INTO warehouses VALUES (ware_id1, ...); EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
ROLLBACK TO do_ins;
END ;
```

При выполнении ROLLBACK TO все точки сохранения, отмеченные после указанной, стираются, а все изменения, сделанные после этой точки, отменяются. Однако сама точка сохранения, к которой вы возвращаетесь, не удаляется. Например, если вы последовательно отметите точки сохранения A, B C и D, а затем выполните ROLLBACK TO к точке B, то будут стерты лишь C и D. ROLLBACK без аргументов, как и COMMIT, стирает все точки сохранения. Если вы отмечаете точку сохранения в рекурсивной подпрограмме, то на каждом уровне рекурсивного спуска предложение SAVEPOINT будет создавать новые экземпляры точек сохранения. Однако ROLLBACK TO вернет вас лишь к самой последней из точек сохранения с данным именем. Имена точек сохранения - это необъявляемые идентификаторы. Их можно повторно использовать внутри транзакции. По умолчанию число активных точек сохранения на сессию не может быть больше 5. АКТИВНАЯ ТОЧКА СОХРАНЕНИЯ - это точка, отмеченная после последней операции COMMIT или ROLLBACK. Вы или ваш АБД можете поднять этот лимит (вплоть до 255), увеличив значение параметра инициализации ORACLE с именем SAVEPOINTS.

Неявные точки сохранения

Перед выполнением каждого предложения INSERT, UPDATE и DELETE ORACLE создает неявную точку сохранения (недоступную вам). Если предложение сбивается, то выполняется откат к этой неявной точке. Обычно отменяется лишь сбившееся предложение SQL, а не вся транзакция. Однако, если это предложение возбудило необрабатываемое исключение, то хост-окружение определяет, что следует отменить. Если хранимая подпрограмма сбилась в результате необрабатываемого исключения, то ORACLE выполняет неявный откат всех изменений, сделанных этой подпрограммой. Однако, если подпрограмма выдала COMMIT до возникновения необрабатываемого исключения, то отменяется лишь неподтвержденная часть работы.

Завершение транзакций

Хорошей практикой программирования является явное подтверждение или явный откат каждой транзакции. Выдаете ли вы COMMIT и ROLLBACK в вашей программе PL/SQL или в хост-окружении - зависит от логического потока вашего приложения. Если вы пренебрегаете явными операциями COMMIT или ROLLBACK, то окончательное состояние транзакции определяет хост-окружение. Например, в среде SQL*Plus, если ваш блок PL/SQL не содержит предложения COMMIT или ROLLBACK, окончательное состояние вашей транзакции зависит от того, что вы делаете после выполнения этого блока. Если вы выполняете предложение определения данных, предложение управления данными или предложение COMMIT, либо если вы выдаете команду EXIT, DISCONNECT, то ORACLE неявно подтверждает вашу транзакцию. Если вы выдаете предложение ROLLBACK или аварийно снимаете сессию SQL*Plus, то ORACLE выполняет откат транзакции.

Использование SET TRANSACTION

Умалчиваемым режимом для всех транзакций является согласованность данных по чтению НА УРОВНЕ ПРЕДЛОЖЕНИЯ. Это гарантирует, что запрос видит лишь то состояние данных, которое было подтверждено перед началом его выполнения, плюс все изменения, которые внесены предыдущими предложениями в текущей транзакции. Если во время запроса другие пользователи вносят изменения в эти же таблицы базы данных, то эти изменения будут видны лишь последующим, но не текущему, запросу. Однако вы можете, выдав предложение SET TRANSACTION, установить режим согласованности данных по чтению НА УРОВНЕ ТРАНЗАКЦИИ. Это гарантирует, что запрос видит лишь то состояние данных, которое было подтверждено перед началом всей транзакции; однако при этом транзакция не должна вносить изменений в базу данных. Предложение SET TRANSACTION READ ONLY не принимает дополнительных параметров и имеет вид: SET TRANSACTION READ ONLY; Предложение SET TRANSACTION должно быть первым предложением SQL в транзакции и может появиться лишь один раз на транзакцию. Как уже сказано, в таком режиме транзакции все запросы, выдаваемые в ней, видят то состояние данных, которое было подтверждено перед началом всей транзакции. Режим READ ONLY не влияет на других пользователей или другие транзакции. В транзакции READ ONLY допускаются лишь предложения SELECT, COMMIT и ROLLBACK. Другие предложения, например, INSERT или DELETE, приводят к возбуждению исключения. В течение транзакции READ ONLY все ее запросы обращаются к одному и тому же снимку базы данных,

что обеспечивает многотабличное, многозапросное, согласованное по чтению представление данных для транзакции. Другие пользователи могут продолжать опрашивать или обновлять данные в обычном режиме. Транзакция READ ONLY завершается выдачей COMMIT или ROLLBACK. В следующем примере вы, как управляющий складом, используете транзакцию READ ONLY, чтобы собрать цифры по продажам за день, прошедшую неделю и прошедший месяц. На эти цифры не могут повлиять другие пользователи, обновляющие базу данных во время транзакции.

Синтаксис операторов SQL, определяющих уровни изоляции

Уровень изоляции транзакции задается следующим оператором:

```
SET TRANSACTION {ISOLATION LEVEL
{READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE}
| {READ ONLY | READ WRITE}}.,..
```

Этот оператор определяет режим выполнения следующей транзакции, т.е. этот оператор не влияет на изменение режима той транзакции, в которой он подается. Обычно, выполнение оператора SET TRANSACTION выделяется как отдельная транзакция:

```
… (предыдущая транзакция выполняется со своим уровнем изоляции)

COMMIT;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
COMMIT;

… (следующая транзакция выполняется с уровнем изоляции REPEATABLE READ)
```

Если задано предложение ISOLATION LEVEL, то за ним должно следовать один из параметров, определяющих уровень изоляции.

Кроме того, можно задать признаки READ ONLY или READ WRITE. Если указан признак READ ONLY, то предполагается, что транзакция будет только читать данные. При попытке записи для такой транзакции будет сгенерирована ошибка. Признак READ ONLY введен для того, чтобы дать производителям СУБД возможность уменьшать количество блокировок путем использования других методов сериализации (например, метод выделения версий).

Оператор SET TRANSACTION должен удовлетворять следующим условиям:

- Если предложение ISOLATION LEVEL отсутствует, то по умолчанию принимается уровень SERIALIZABLE.

- Если задан признак READ WRITE, то параметр ISOLATION LEVEL не может принимать

значение READ UNCOMMITTED.

- Если параметр ISOLATION LEVEL определен как READ UNCOMMITTED, то транзакция становится по умолчанию READ ONLY. В противном случае по умолчанию транзакция считается как READ WRITE.

Переопределение умалчиваемой блокировки

По умолчанию ORACLE автоматически блокирует для вас структуры данных. Однако вы можете запросить специфические блокировки по строкам или таблицам, если вам почему-либо выгодно изменить умалчиваемый режим блокировки. Явная блокировка позволяет вам разрешать или запрещать совместный доступ к таблице на время транзакции.

Использование FOR UPDATE

При объявлении курсора (работу с курсорами см. ниже), к которому будет обращаться фраза ***WHERE CURRENT OF*** предложения UPDATE или DELETE, вы должны использовать фразу FOR UPDATE, чтобы затребовать для этого курсора монопольные блокировки строк. Фраза FOR UPDATE, когда она присутствует, должна появляться в конце объявления курсора, как показывает следующий пример:

```
DECLARE

CURSOR c1 IS SELECT volume FROM goods WHERE goods_id = 5 FOR UPDATE;
```

Фраза ***FOR UPDATE*** указывает, что строки, выбираемые запросом, будут обновляться или удаляться, и блокирует все строки в активном множестве курсора. Это полезно, когда вы хотите, чтобы обновление базировалось на существующих значениях строк. В этом случае вам нужна гарантия, что строка не будет изменена другим пользователем, прежде чем вы обновите ее. Все строки в активном множестве блокируются в момент открытия курсора, и разблокируются при выполнении COMMIT. Поэтому после COMMIT вы не можете извлекать строк из курсора, объявленного FOR UPDATE. При запросах по нескольким таблицам вы можете использовать фразу FOR UPDATE OF, чтобы ограничить блокировку строк конкретными таблицами. Строки будут блокироваться лишь в тех таблицах, которые содержат столбец, указанный в фразе FOR UPDATE OF.

Например, следующий запрос блокирует строки в таблице storages , но не в таблице warehouses :

```
DECLARE
CURSOR stor_goods IS
```

```
SELECT storages.quantity FROM storages WHERE
Storages.ware_id= warehouses.ware_id AND Address='KHARKOV'
FOR UPDATE OF quantity;
```

Чтобы обратиться к последней строке, извлеченной курсором, вы используете фразу WHERE CURRENT OF предложения UPDATE или DELETE , как показывает следующий пример:

```
UPDATE storages SET quantity = 0 WHERE CURRENT OF stor_goods;
```

Использование LOCK TABLE

Предложение ***LOCK TABLE*** позволяет вам заблокировать одну или несколько таблиц в указанном режиме, так что вы можете регулировать одновременный доступ к таблицам, поддерживая их целостность.

Например, предложение, приведенное ниже, блокирует таблицу goods в режиме row share. Такой режим разрешает одновременный доступ к таблице, но запрещает другим пользователям блокировать всю таблицу для монопольного использования. Блокировка таблицы освобождается, когда ваша транзакция выдает COMMIT или ROLLBACK.

Пример:

```
LOCK TABLE goods IN ROW SHARE MODE NOWAIT;
```

Режим блокировки определяет, какие другие блокировки могут быть применены к таблице. Например, несколько пользователей могут одновременно затребовать блокировки row share для одной и той же таблицы, но лишь один пользователь за раз может затребовать МОНОПОЛЬНУЮ (exclusive) блокировку. Пока один пользователь имеет монопольную блокировку таблицы, другие пользователи не могут изменять (INSERT, UPDATE или DELETE) строк в этой таблице.

Необязательное ключевое слово ***NOWAIT*** указывает, что, если запрос LOCK TABLE не может быть удовлетворен (возможно, потому, что таблица уже заблокирована другим пользователем), то LOCK TABLE вернет управление пользователю, вместо того, чтобы ждать удовлетворения запроса. Если вы опустите ключевое слово NOWAIT, то ORACLE будет ждать освобождения таблицы; это ожидание не имеет устанавливаемого предела. Блокировка таблицы никогда не препятствует другим пользователям выдавать запросы по этой таблице; с другой стороны, запрос никогда не требует блокировки таблицы. Одной транзакции придется ждать завершения другой лишь в том случае, если эти транзакции пытаются модифицировать одну и ту же строку.

Восстановление данных. Управление параллелизмом

Виды восстановления данных

Восстановление базы данных может производиться в следующих случаях:

- ***Индивидуальный откат транзакции***. Откат индивидуальной транзакции может быть инициирован либо самой транзакцией путем подачи команды ROLLBACK, либо системой. СУБД может инициировать откат транзакции в случае возникновения какой-либо ошибки в работе транзакции (например, деление на нуль) или если эта транзакция выбрана в качестве жертвы при разрешении тупика.

- ***Мягкий сбой системы*** (***аварийный отказ программного обеспечения***). Мягкий сбой характеризуется утратой оперативной памяти системы. При этом поражаются все выполняющиеся в момент сбоя транзакции, теряется содержимое всех буферов базы данных. Данные, хранящиеся на диске, остаются неповрежденными. Мягкий сбой может произойти, например, в результате аварийного отключения электрического питания или в результате неустранимого сбоя процессора.

- ***Жесткий сбой системы*** (***аварийный отказ аппаратуры***). Жесткий сбой характеризуется повреждением внешних носителей памяти. Жесткий сбой может произойти, например, в результате поломки головок дисковых накопителей.

Во всех трех случаях основой восстановления является избыточность данных, обеспечиваемая журналом транзакций.

Как и страницы базы данных, данные из журнала транзакций не записываются сразу на диск, а предварительно буферизируются в оперативной памяти. Таким образом, система поддерживает два вида буферов - буферы страниц базы данных и буферы журнала транзакций.

Страницы базы данных, содержимое которых в буфере (в оперативной памяти) отличается от содержимого на диске, называются ***"грязными" (dirty) страницами***. Система постоянно поддерживает список "грязных" страниц - ***dirty-список***. Запись "грязных" страниц из буфера на диск называется ***выталкиванием страниц во внешнюю память***. Очевидно, необходимо предусмотреть такие правила выталкивания буферов базы данных и буферов журнала транзакций, которые обеспечивали бы два требования:

- *Максимальную скорость выполнения транзакций*. Для этого необходимо выталкивать страницы как можно реже. В идеале, если оперативная память была бы бесконечной, и сбои никогда бы не происходили, наилучшим выходом была бы загрузка *всей* базы данных в оперативную память, работа с данными *только в оперативной памяти*, и запись измененных страниц на диск *только* в момент завершения работы всей системы.

- Гарантию, что при возникновении сбоя (любого типа), данные завершенных транзакций можно было бы восстановить, а данные незавершенных транзакций бесследно удалить, т.е. обеспечение восстановления последнего согласованного состояния базы данных. Для этого что-то выталкивать на диск все-таки необходимо, даже если мы обладали бы бесконечной оперативной памятью.

Таким образом, имеется две причины для периодического выталкивания страниц во внешнюю память - недостаток оперативной памяти и возможность сбоев.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц базы данных является то, что запись об изменении объекта базы данных должна попадать во внешнюю память журнала раньше, чем измененный объект оказывается во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется ***Write Ahead Log*** (***WAL***) - "***пиши сначала в журнал***", и состоит в том, что если требуется вытолкнуть во внешнюю память измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала записи о его изменении. Это означает, что если во внешней памяти базы данных содержится объект, к которому применена некоторая команда модификации, то во внешней памяти журнала транзакций содержится запись об этой операции. Обратное неверно - если во внешней памяти журнала содержится запись о некотором изменении объекта, то во внешней памяти базы данных может и не быть самого измененного объекта.

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

Третьим условием выталкивания буферов является ограниченность объемов буферов базы данных и журнала транзакций. Периодически или при наступлении определенного события (например, количество страниц в dirty-списке превысило определенный порог, или количество свободных страниц в буфере уменьшилось и достигло критического значения) система принимает так называемую ***контрольную точку***. Принятие контрольной точки включает выталкивание во внешнюю память содержимого буферов базы данных и специальную физическую ***запись контрольной точки***, которая представляет собой список всех осуществляемых в данный момент транзакций.

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

Индивидуальный откат транзакции

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

Индивидуальный откат транзакции выполняется следующим образом:

а) Просматривается список записей, сделанных данной транзакцией в журнале транзакций (от последнего изменения к первому изменению).

б) Выбирается очередная запись из списка данной транзакции.

в) Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT, и вместо прямой операции UPDATE обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.

г) Любая из этих обратных операций также журнализируются. Это необходимо делать, потому что во время выполнения индивидуального отката может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.

д)При успешном завершении отката в журнал заносится запись о конце транзакции.

Восстановление после мягкого сбоя

Несмотря на протокол WAL, после мягкого сбоя не все физические страницы базы данных содержат измененные данные, т.к. не все "грязные" страницы базы данных были вытолкнуты во внешнюю память.

Последний момент, когда гарантированно были вытолкнуты "грязные" страницы - это момент принятия последней контрольной точки. Имеется 5 вариантов состояния транзакций по отношению к моменту последней контрольной точки и к моменту сбоя (рис.1) .

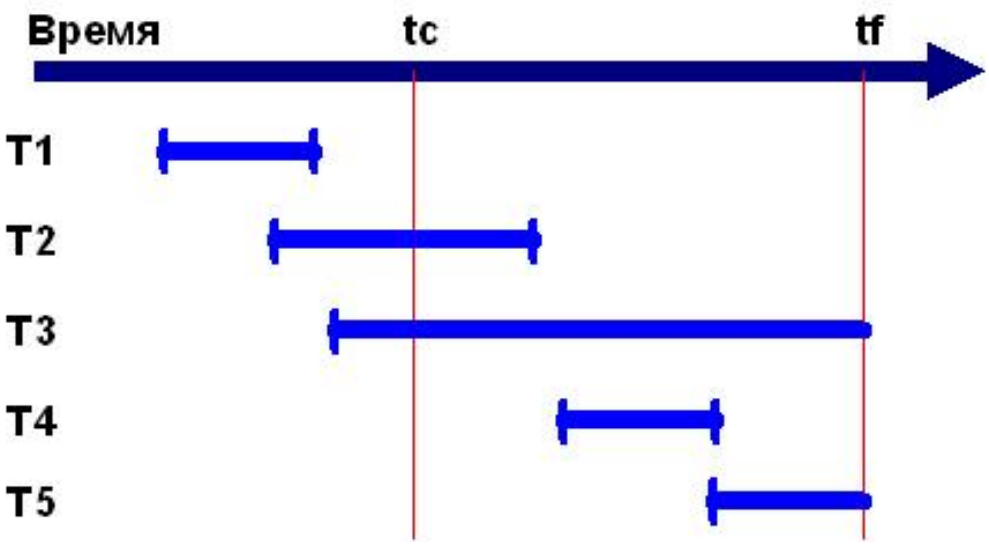

Рисунок 1- Пять вариантов транзакций (tc - контрольная точка, tf - отказ системы)

Последняя контрольная точка принималась в момент tc. Мягкий сбой системы произошел в момент tf. Транзакции T1-T5 характеризуются следующими свойствами.

▼ **Подробнее**

T1 - транзакция успешно завершена до принятия контрольной точки. Все данные этой транзакции сохранены в долговременной памяти - как записи журнала, так и страницы данных, измененные этой транзакцией. Для транзакции T1 никаких операций по восстановлению не требуется.

T2 - транзакция начата до принятия контрольной точки и успешно завершена после контрольной точки, но до наступления сбоя. Записи журнала транзакций, относящиеся к этой транзакции вытолкнуты во внешнюю память. Страницы данных, измененные этой транзакцией, только частично вытолкнуты во внешнюю память. Для данной транзакции необходимо повторить заново те операции, которые были выполнены после принятия контрольной точки.

T3 - транзакция начата до принятия контрольной точки и не завершена в результате сбоя. Такую транзакцию необходимо откатить. Проблема, однако, в том, что часть страниц данных, измененных этой транзакцией, уже содержится во внешней памяти - это те страницы, которые были обновлены до принятия контрольной точки. Следов изменений, внесенных после контрольной точки в базе данных нет. Записи журнала транзакций, сделанные до принятия контрольной точки, вытолкнуты во внешнюю память, те записи журнала, которые были сделаны после контрольной точки, отсутствуют во внешней памяти журнала.

T4 - транзакция начата после принятия контрольной точки и успешно завершена до сбоя системы. Записи журнала транзакций, относящиеся к этой транзакции вытолкнуты во внешнюю память журнала. Изменения в базе данных, внесенные этой транзакцией, полностью отсутствуют во внешней памяти базы данных. Такую транзакцию необходимо повторить целиком.

T5 - транзакция начата после принятия контрольной точки и не завершена в результате сбоя. Никаких следов этой транзакции нет ни во внешней памяти журнала транзакций, ни во внешней памяти базы данных. Для такой транзакции никаких действий предпринимать не нужно, ее как бы и не было вовсе.

Восстановление системы после мягкого сбоя осуществляется как часть процедуры перезагрузки системы. При перезагрузке системы транзакции T2 и T4 необходимо частично или полностью повторить, транзакцию T3 - частично откатить, для транзакций T1 и T5 никаких действий предпринимать не нужно. При перезагрузке система выполняет следующие действия:

- Создается два списка транзакций UNDO (отменить) и REDO (повторить). В список UNDO заносятся все транзакции из последней записи контрольной точки (т.е. все транзакции, выполнявшиеся в момент принятия контрольной точки). Список REDO остается пустым. В нашем случае будет: UNDO = {T2, T3}, REDO = { }.

- Начиная с записи контрольной точки просматривается вперед журнал транзакций.

- Если в журнале транзакций обнаруживается запись о начале транзакции, то эта транзакция добавляется в список UNDO. В нашем случае будет: UNDO = {T2, T3, T4}, REDO = { }. Заметим, что следов транзакции T5 в журнале транзакций нет.

- Если в файле регистрации обнаруживается запись COMMIT об окончании транзакции, то эта транзакция добавляется в список REDO. В нашем случае будет: UNDO = {T2, T3, T4}, REDO = {T2, T4}. Заметим, что записи о конце этих транзакций имеются во внешней памяти журнала транзакций в соответствии с минимальным требованием выталкивания записей журнала при фиксации транзакции.

- Когда достигается конец журнала транзакций, оба списка анализируются. При этом из списка UNDO удаляются те транзакции, которые попали в список REDO. В нашем случае будет: UNDO = {T3}, REDO = {T2, T4}.

- После этого система просматривает журнал транзакций назад, начиная с момента контрольной точки и откатывая все транзакции из списка UNDO. В нашем случае будут откатываться те операции транзакции T3, которые были выполнены до принятия контрольной точки.

- Окончательно, система просматривает журнал транзакций вперед, начиная с момента контрольной точки, и повторно выполняет все операции транзакций из списка REDO. В нашем случае, система выполнит повторно все операции транзакции T4 и те операции транзакции T2, которые были выполнены после принятия контрольной точки.

Восстановление после жесткого сбоя

При жестком сбое база данных на диске нарушается физически. Основой восстановления в этом случае является журнал транзакций и ***архивная копия базы данных***. Архивная копия базы данных должна создаваться периодически, а именно с учетом скорости наполнения журнала транзакций.

Восстановление начинается с обратного копирования базы данных из архивной копии. Затем выполняется просмотр журнала транзакций для выявления всех транзакций, которые закончились *успешно* до наступления сбоя. (Транзакции, закончившиеся откатом до наступления сбоя, можно не рассматривать). После этого по журналу транзакций в прямом направлении повторяются все успешно законченные транзакции. При этом нет необходимости отката транзакций, прерванных в результате сбоя, т.к. изменения, внесенными этими транзакциями, отсутствуют после восстановления базы данных из резервной копии.

Наиболее плохим случаем является ситуация, когда разрушены физически и база данных, и журнал транзакций. В этом случае единственное, что можно сделать - это восстановить состояние базы данных на момент последнего резервного копирования. Для того чтобы не допустить возникновение такой ситуации, базу данных и журнал транзакций обычно располагают на *физически* разных дисках, управляемых физически разными контроллерами.

Восстановление данных и стандарт SQL

Стандарт языка SQL не содержит требований к восстановимости данных, оставляя эти вопросы на усмотрение разработчиков СУБД.

Восстановление и копирование базы данных Oracle

Независимо от того, какую схему копирования и восстановления изберете для базы данных ORACLE, резервное копирование средствами операционной системы всех файлов базы данных абсолютно необходимо как часть общей стратегии предохранения от потенциальных сбоев носителя, которые могут повредить эти файлы.

Полное копирование - это копирование средствами операционной системы всех файлов данных, онлайновых файлов журнала и управляющих файлов, составляющих базу данных ORACLE. Полное копирование должно также включать копирование файлов параметров, ассоциированных с этой базой данных. Полное копирование базы данных проводится, когда база данных чисто закрыта; это означает, что оно не проводится после сбоя инстанции и т.п. В это время все файлы, составляющие базу данных, закрыты, и содержимое их согласовано по отношению к текущему моменту времени.

Копии файлов данных, полученные при полном копировании, полезны при любом типе восстановления носителя:

- если база данных работает в режиме NOARCHIVELOG, и сбой диска повреждает некоторые или все файлы, составляющие базу данных, то самые последние полные копии могут использоваться для *реставрации* (не восстановления) базы данных. Так как архивированный

журнал повторения недоступен для приведения базы данных в состояние на текущий момент, вся работа, проделанная в базе данных после последнего полного копирования, должна быть повторена заново;

- если база данных работает в режиме ARCHIVELOG, и сбой диска повреждает некоторые или все файлы, составляющие базу данных, то самые последние полные копии могут использоваться как часть *восстановления* базы данных. После реставрации необходимых файлов данных из полных копий можно продолжить восстановление путем применения архивированных и текущих онлайновых файлов журнала повторения, чтобы привести реставрированные файлы в состояние на текущий момент времени.

Частичное копирование - это любое (отличное от полного) копирование средствами операционной системы файлов базы данных, осуществляемое, когда база данных открыта либо закрыта. Примерами частичного копирования служат:

- копирование всех файлов данных конкретного табличного пространства;

- копирование одиночного файла данных;

- копирование управляющего файла.

Частичные копии могут быть полезными лишь для базы данных, работающей в режиме ARCHIVELOG. Так как архивированный журнал повторения присутствует, файлы данных, реставрированные из частичных копий, могут быть согласованы с остальной частью базы данных с помощью процедур восстановления.

Частичное копирование может включать файлы данных базы данных. Отдельные файлы данных или их группы можно копировать независимо от остальных файлов данных, онлайновых файлов журнала и управляющих файлов. Файл данных можно копировать, когда он в состоянии офлайн или онлайн.

Любой файл данных можно копировать, когда он находится в *офлайне.* Следующие ситуации показывают примеры офлайновых копирований:

- БД закрыта. Как следствие, все файлы данных БД закрыты, или находятся в состоянии офлайн. Любое копирование файлов данных закрытой БД считается офлайновым копированием.

- БД открыта, но табличное пространство находится в состоянии офлайн. Как следствие, все файлы данных этого табличного пространства обычно находятся в офлайне. Любое копирование файлов данных офлайнового табличного пространства считается офлайновым копированием.

Если база данных работает в режиме ARCHIVELOG, то любой файл данных можно копировать, когда база данных открыта, ассоциированное табличное пространство в онлайне, и сам этот файл данных находится в онлайне и подвергается нормальным операциям базы данных. Такой тип копирования называется ***онлайновым копированием***.

Копия онлайнового файла данных является копией *несогласованных* данных; нельзя

гарантировать, что все данные в этой копии согласованы по отношению к какой-либо точке времени. Однако эти данные легко сделать согласованными с помощью процедур восстановления базы данных.

Когда начинается копирование онлайнового табличного пространства или индивидуального файла данных (командой ALTER TABLESPACE с опцией BEGIN BACKUP), ORACLE временно прекращает запись контрольных точек в заголовки копируемых файлов данных. Это значит, что после реставрации этого файла данных из копии он "знает" лишь о самой последней контрольной точке, которая была выполнена ПЕРЕД копированием табличного пространства, но не о тех контрольных точках, которые происходили ВО ВРЕМЯ копирования. Как следствие, ORACLE требует, чтобы по время восстановления были применены соответствующие файлы журнала повторения. Когда онлайновое копирование заканчивается (как указывается командой ALTER TABLESPACE с опцией END BACKUP), ORACLE продвигает заголовок файла данных к текущей контрольной точке базы данных.

Восстановление базы данных

Процедуры восстановления

В любой системе баз данных вероятность сбоя системы существует всегда. Когда происходит сбой системы, база данных должна быть восстановлена как можно быстрее и с минимальным возможным ущербом для пользователей.

Восстановление от любого типа системного сбоя требует:

- Определения, какие структуры базы данных не затронуты, а какие требуют восстанвления.

- Осуществления требуемых шагов восстановления.

- Перезапуска базы данных для возобновления нормальной работы.

- Проверки, что никакая работа не потеряна, и что в базе данных нет некорректных данных.

Цель - как можно скорее вернуться к норме, и в то же время изолировать пользователей базы данных от любых проблем и защитить их от возможной потери или дублирования их работы.

Процесс восстановления варьируется в зависимости от типа сбоя и от того, какие файлы базы данных затронуты этим сбоем.

Многовариантность

Эта тема очень тесно связана с управлением одновременным доступом, поскольку создает

основу для механизмов управления одновременным доступом в СУБД Oracle - Oracle использует модель многовариантной согласованности по чтению при одновременном доступе. По сути это механизм, с помощью которого СУБД Oracle обеспечивает:

- согласованность по чтению для запросов: запросы выдают согласованные результаты на момент начала их выполнения;

- неблокируемые запросы: запросы не блокируются сеансами, в которых изменяются данные, как это бывает в других СУБД.

Это две очень важные концепции СУБД Oracle. Термин многовариантность произошел от того, что фактически СУБД Oracle может одновременно поддерживать множество версий данных в базе данных. Понимая сущность многовариантности, всегда можно понять результаты, получаемые из базы данных.

Еще одна демонстрация многовариантности: в БД имеется несколько версий одной и той же информации, по состоянию на различные моменты времени. СУБД Oracle использует эти сделанные в разное время "моментальные снимки" данных для поддержки согласованности по чтению и неблокируемости запросов. Это согласованное по чтению представление данных всегда выполняется на уровне оператора SQL, - результаты выполнения любого оператора SQL всегда согласованы на момент его начала. Именно это свойство позволяет получать предсказуемый набор данных в результате.

Реализация блокирования

СУБД использует блокировки, чтобы в каждый момент времени те или иные данные могли изменяться только одной транзакцией. Говоря проще, блокировки - это механизм обеспечения одновременного доступа. **При** отсутствии определенной модели блокирования, предотвращающей одновременное изменение, например, одной строки, многопользовательский доступ к базе данных попросту невозможен. Однако при избыточном или неправильном блокировании одновременный доступ тоже может оказаться невозможным. Если пользователь или сама СУБД блокирует данные без необходимости, то работать одновременно сможет меньшее количество пользователей. Поэтому понимание назначения блокирования и способов его реализации в используемой СУБД принципиально важно для создания корректных и масштабируемых приложений.

Блокировка - механизм SQL для управления параллельными операциями. Блокировки приостанавливают определенные операции над БД на то время, пока активны другие операции или транзакции.

Разделяемые блокировки (shared locks) или S-блокировки (S-locks) - могут одновременно устанавливаться многими пользователями, т.е. любой пользователь имеет доступ к данным, но не может изменять их.

Исключительные блокировки (exclusive locks) или X-блокировки (X-locks) - позволяют

иметь доступ к данным только владельцу блокировки.

Последовательность установления блокировок Oracle

- нахождение адреса строки, которую необходимо заблокировать;

- переход на эту строку;

- блокировка этой строки (ожидая снятия блокировки, если она уже заблокирована и при этом на используется NOWAIT).

Простейшими средствами обеспечения очередности доступа, используемые для координации многопользовательского доступа к общим структурам данных, объектам и файлам являются внутренние блокировки и фиксаторы.

Фиксаторы - блокировки, удерживаемые в течении непродолжительного времени (например для изменения структуры данных в памяти). Используются для защиты определенных структур памяти.

Один процесс - один фиксатор. Если владелец фиксатора «скончается», то очистка выполняется процессом PMON.

Внутренние блокировки можно устанавливать на разных уровнях, что позволяет иметь несколько разделяемых блокировок и блокировать с разными уровнями совместимости. Работают медленнее фиксаторов, но обеспечивают большие функциональные возможности.

Пессимистические блокировки (pessimistic locks) - предотвращают доступ к данным для одновременных транзакций.

Этот метод блокирования должен использоваться непосредственно перед изменением значения на экране, например, когда пользователь выбирает определенную строку с целью изменения (допустим, щелкая на кнопке в окне). Итак, пользователь запрашивает данные без блокирования:

Приложение передает значения для связываемых переменных в соответствии с данными на экране и повторно запрашивает ту же самую строку из базы данных, но в этот раз блокирует ее изменения другими сеансами. Вот почему такой подход называется *пессимистическим* блокированием. Мы блокируем строку перед попыткой изменения, поскольку сомневается, что она останется неизменной.

Оптимистические блокировки (optimistic locks) - отслеживают возникновение конфликтов и при необходимости выполняют откат транзакций.

Этот метод состоит в том, чтобы сохранять старое и новое значения в приложении и использовать их при изменении следующим образом:

Update table

Set column1 = :new_column1, column2 = :new_column2, ...

Where column1 = :old_column1

And column2 = :old_column2

Здесь мы оптимистически надеемся, что данные не изменились. Если в результате изменена **одна** строка, значит, нам повезло: данные не изменились с момента считывания. Если изменено **ноль** строк, мы проиграли - кто-то уже изменил данные и необходимо решить, что делать, чтобы это изменение не потерять.

СУБД Oracle обеспечивает грануляцию блокировок по уровням (см. табл. 17)

Таблица 17 - Грануляция блокировок

Уровни блокирования	Происходящее
Таблица	блокируется вся таблица
Пространство таблицы или БД	блокируется физическая область накопителя, в которой размещаются как часть таблицы, так и несколько таблиц
Строки	применяется к определенной строке таблицы
Страницы	блокирование страницы (блока) данных на физическом уровне, *(очень эффективна с точки зрения производительности)*
Элемент	применяется только к одному значению, столбцу или строке (идеально с точки зрения параллелизма, но работает очень медленно)

Эскалация блокировок - увеличение размера блокируемых объектов.

В Oracle применяется ***преобразование блокировок*** - блокировка низкого уровня преобразуется к более высокому (ограничивающему) уровню.

Стоит заметить, что и в этом случае тоже можно использовать оператор **SELECT FOR UPDATE NOWAIT.** Представленный выше оператор **UPDATE** позволяет избежать потери изменений, но может приводить к блокированию, "зависая" в ожидании завершения изменения строки другим сеансом. Если все приложения используют оптимистическое блокирование, то применение простых операторов UPDATE вполне допустимо, поскольку строки блокируются на очень короткое время выполнения и фиксации изменений. Однако если некоторые приложения используют пессимистическое блокирование, удерживая блокировки строк достаточно долго, имеет смысл выполнять оператор **SELECT FOR UPDATE NOWAIT** непосредственно перед оператором **UPDATE,** чтобы избежать блокирования другим сеансом

Пример:

Select … FOR UPDATE; => исключительная блокировка выбранных строк и ROW SHARED TABLE - совместное блокирование строк таблицы (чтобы ни один сеанс не могли заблокировать всю

таблицу). После изменения - преобразование блокировок.

При использовании пессимистического блокирования пользователь может быть уверен, что изменяемые им на экране данные сейчас ему "принадлежат" - он получил запись в свое распоряжение, и никто другой не может ее изменять. Можно возразить, что, блокируя строку до изменения, вы лишаете к ней доступа других пользователей и, тем самым, существенно снижаете масштабируемость приложения. Но обновлять строку в каждый момент времени сможет только один пользователь (если мы не хотим потерять изменения). Если сначала заблокировать строку, а затем изменять ее, пользователю будет удобнее работать. Если же пытаться изменить, не заблокировав заранее, пользователь может напрасно потерять время и силы на изменения, чтобы в конечном итоге получить сообщение: "Извините, данные изменились, попробуйте еще раз". Чтобы ограничить время блокирования строки перед изменением, можно снимать блокировку в приложении, если пользователь занялся чем-то другим и некоторое время не использует строку, или использовать профили ресурсов (Resource Profiles) в базе данных для отключения простаивающих сеансов.

Более того, блокирование строки в Oracle не мешает ее читать, как в других СУБД; блокирование строки не мешает обычной работе с базой данных. Все это исключительно благодаря соответствующей реализации механизмов одновременного доступа и блокирования в Oracle. В других СУБД верно как раз обратное. Если попытаться использовать в них пессимистическое блокирование, ни одно приложение не будет работать. Тот факт, что в этих СУБД блокирование строки не дает возможности выполнять к ней запросы, не позволяет даже рассматривать подобный подход. Поэтому иногда приходится "забывать" правила, выработанные в процессе работе с одной СУБД, чтобы успешно разрабатывать приложения для другой.

Типы блокировок

Выделяют общие блокировки, к которым относятся блокировки ЯМД (DML Locks), блокировки ЯОД (DDL Locks) и внутренние блокировки и фиксаторы (защелки) (internal locks, latches), а также для OPS - Oracle parallel Server характерны распределенные блокировки (distributed locks), которые используются для согласования ресурсов машин, входящих в кластер р (устанавливаются экземплярами баз данных, а не отдельными транзакциями) и блокировки параллельного управления кэшем (PCM Parallel Cache Management Locks) - защищают блоки данных в кэше при использовании несколькими экземплярами.

Подробнее рассмотрим общие блокировки.

▼ **Подробнее**

Блокировки ЯМД (DML Locks) позволяют гарантировать, что в каждый момент времени только одному сеансу позволено изменять строку и что не может быть удалена таблица, с которой работает сеанс. Установка блокировок - автоматически по ходу работы.

Выделяют TX - блокировки транзакций и TM - блокировки очередности ЯМД.

TX-блокировка устанавливается, когда транзакция инициирует первое изменение и удерживается до выполнения Commit или Rollback. Используется как механизм организации очереди для сеансов, ожидающих завершения транзакции.

Информация о блокировках хранится как часть служебной информации блока - управляется атрибутами хранения INITRANS, MAXTRANS (можно посмотреть в V$TRANSACTION, V$SESSION, V$LOCK).

При установлении TM-блокировки структура таблицы не может быть изменена при изменении ее содержимого. Общее количество TM - блокировок конфигурируется администратором (параметр DBMS_Lokcs в файле init.ora). Если 0, то не разрешены операции DDL.

Ниже приведенной командой устанавливается TM-блокировка:

Alter Table имя_таблицы DISABLE TABLE LOCK.

Блокировки ЯОД автоматически устанавливаются на объекты в ходе выполнения операторов DDL для защиты их от изменения другими сеансами. Операторы DDL всегда фиксируют транзакцию (даже при неудачном завершении).

Выделяют исключительные блокировки, разделяемые блокировки и нарушаемые блокировки разбора.

Исключительные блокировки:

- Предотвращают установку блокировок DDL или TM другими сеансами.

- Можно запрашивать объекты в ходе выполнения операторов DDL, но нельзя его изменять.

- Большинство операторов DDL устанавливают Исключительную блокировку DDL .

Разделяемые блокировки:

- Защищают структуру соответствующего объекта от изменения другими сеансами, но разрешают изменять данные.

- Устанавливаются на объекты от которых зависят скомпилированные хранимые объекты, типы процедур и представлений.

Нарушаемые блокировки разбора:

- Позволяют объекту (план запроса, хранящемуся в кэше разделяемого пула) зарегистрировать свою зависимость от другого объекта.

- Используется для отметки «недействительный» при изменении или удалении объекта, упоминаемого в операторе.

- Они не предотвращают выполнение операторов DDL.

Примеры блокирования вручную

SELECT … FOR UPDATE - основной метод явного блокирования данных вручную.

LOCK TABLE - блокировка таблиц, но не строк в ней.

LOCK TABLE IN EXCLUSIVE MODE - блокировка таблицы в эксклюзивном режиме (используется редко, при выполнении большого пакетного изменения).

Уровни изоляции транзакций

Уровни изоляции транзакций показывают какие типы конфликтов допустимы (см. табл.18)

Таблица 18 - Уровни изволяции транзакций

Уровень изоляции	Потерянные изменения	Грязное чтение	Неповторяющееся чтение	Фантом
READ UNCOMMITED	Нет	Да	Да	Да
READ COMMITED	Нет	Нет	Да	Да
REPEATABLE READ	Нет	Нет	Нет	Да
SERIALIZABLE	Нет	Нет	Нет	Нет

READ UNCOMMITED (чтение без фиксации) - допускает неоднократное выполнение одного и того же запроса с разными результатами независимо от того, были ли результаты параллельных транзакций зафиксированы.

READ COMMITED (чтение с фиксацией) - допускает неоднократное выполнение одного и того же запроса с разными результатами, но при условии, что результаты параллельных транзакций были зафиксированы.

REPEATABLE READ (повторяющееся чтение) - допускает из всех возможных видов неповторяющегося чтения только фантомные вставки.

SERIALIZABLE (последовательное выполнение) - каждая транзакция выполняется изолированно, не влияя на выполнение других параллельных транзакций.

На всех уровнях изоляции потерянные изменения не допускаются, т.е. при наличии внутри транзакции запроса параллельным транзакциям запрещается обновлять или удалять любые данные, выбранные этим запросом, до полного завершения транзакции.

Транзакции только для чтения (***READ ONLY)*** очень похожи на транзакции с уровнем

изолированности **SERIALIZABLE.** Эти транзакция не устанавливает никаких блокировок на данные, но и не может их обновлять.

Транзакция ***READ WRITE*** - может выполнять как запросы, так и изменения данных.

Двухфазная фиксация (2ФФ)

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.

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 "ОК". или, наоборот, - "Not OK".

Во вторых, 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 "ОК", 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.

Распределенные транзакции

Распределенные транзакции обращаются к двум и более узлам и обновляют на них данные.

Основная проблема распределенных транзакций - соблюдение логической целостности данных. Транзакция на всех узлах должна завершиться одинаково: или фиксацией, или откатом.

Выполнение распределенных транзакций осуществляется с помощью специального алгоритма, который называется **двухфазная фиксация**.

Координатор транзакции - узел, который контролирует выполнение этого протокола (обычно, тот узел, который инициирует данную транзакцию).

Остальные узлы, на которых выполняется транзакция, называются ***участниками транзакции***.

На рис.2 приведены схемы взаимодействия участников при поддержке протокола двухфазной

фиксации транзакций для различных схем реализации.

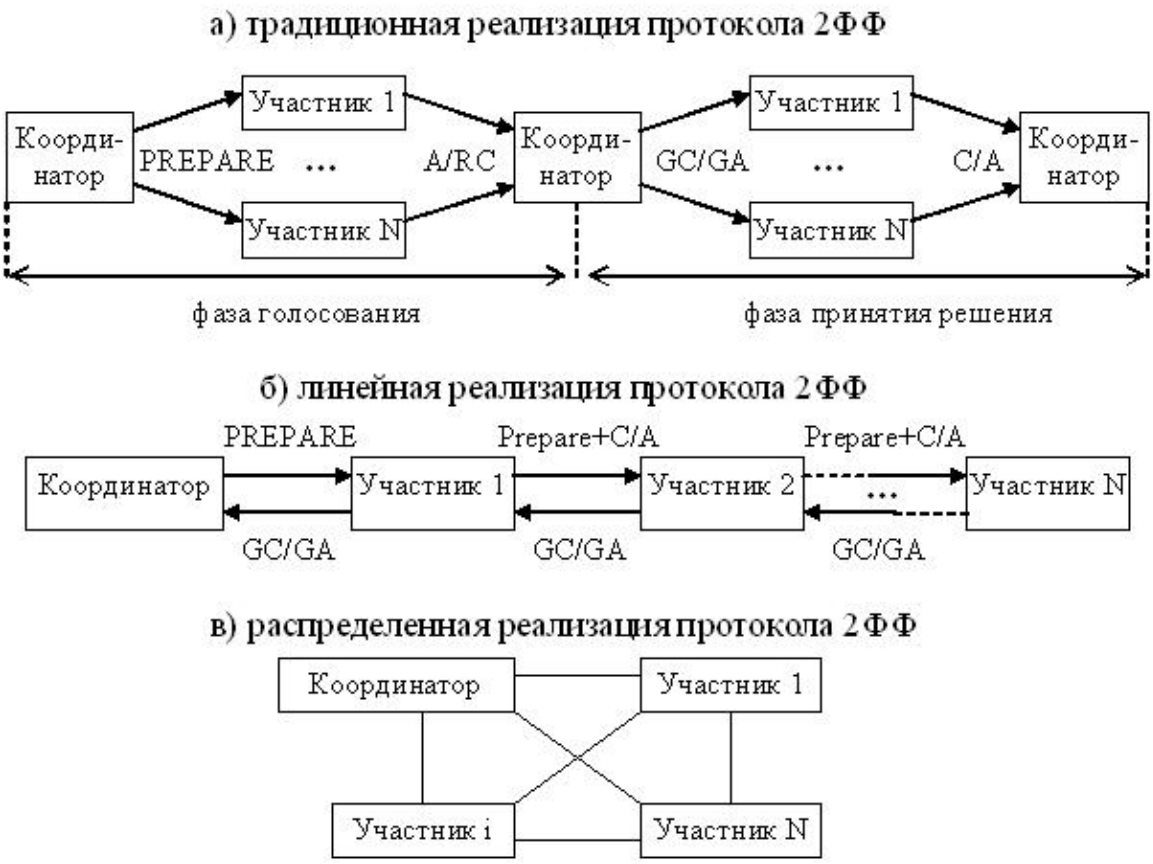

Рисунок 2 - Реализация протокола 2ФФ

На рис. 3 приведен протокол протокол двухфазной фиксации

Рисунок 3 - Протокол двухфазной фиксации

Действия координатора транзакции

▼ **Подробнее**

Координатор выполняет протокол 2ФФ по следующему алгоритму:

Фаза 1 (голосование).

Занести запись begin_commit в системный журнал и обеспечить ее перенос из буфера в ОП на ВЗУ. Отправить всем участникам команду PREPARE.

Ожидать ответов всех участников в пределах установленного тайм-аута.

Фаза 2 (принятие решения).

При поступлении сообщения ABORT: занести в системный журнал запись abort и обеспечить ее перенос из буфера в ОП на ВЗУ; отправить всем участникам сообщение GLOBAL_ABORT и ждать ответов участников (тайм-аут).

Если участник не отвечает в течение установленного тайм-аута, координатор считает, что данный участник откатит свою часть транзакции и запускает протокол ликвидации.

Если все участники прислали COMMIT, поместить в системный журнал запись commit и обеспечить ее перенос из буфера в ОП на ВЗУ. Отправить всем участникам сообщение GLOBAL_COMMIT и ждать ответов всех участников.

После поступления подтверждений о фиксации от всех участников: поместить в системный журнал запись end_transaction и обеспечить ее перенос из буфера в ОП на ВЗУ.

Если некоторые узлы не прислали подтверждения фиксации, координатор заново направляет им сообщения о принятом решении и поступает по этой схеме до получения всех требуемых подтверждений.

Действия участника транзакции

▼ **Подробнее**

Участник выполняет протокол 2ФФ по следующему алгоритму:

а)При получении команды PREPARE, если он готов зафиксировать свою часть транзакции, он помещает запись ready_commit в файл журнала транзакций и отправляет координатору сообщение READY_COMMIT. Если он не может зафиксировать свою часть транзакции, он помещает запись abort в файл журнала транзакций, отправляет координатору сообщение ABORT и откатывает свою часть транзакции (не дожидаясь общего сигнала GLOBAL_ABORT).

б)Если участник отправил координатору сообщение READY_COMMIT, то он ожидает ответа координатора в пределах установленного тайм-аута.

в)При получении GLOBAL_ABORT участник помещает запись abort в файл журнала транзакций, откатывает свою часть транзакции и отправляет координатору подтверждение отката.

г)При получении GLOBAL_COMMIT участник помещает запись commit в файл журнала транзакций, фиксирует свою часть транзакции и отправляет координатору подтверждение фиксации.

д)Если в течение установленного тайм-аута участник не получает сообщения от координатора, он откатывает свою часть транзакции.

Протокол ликвидации для координатора

▼ **Подробнее**

Протокол ликвидации для координатора:

а)Тайм-аут в состоянии WAITING: координатор не может зафиксировать транзакцию, потому что не получены все подтверждения от участников о фиксации. Ликвидация заключается в откате транзакции.

б)Тайм-аут в состоянии DECIDED: координатор повторно рассылает сведения и принятом глобальном решении и ждет ответов от участников.

Простейший протокол ликвидации для участника заключается в блокировании процесса до тех пор, пока сеанс связи с координатором не будет восстановлен. Но в целях повышения производительности (и автономности) узлов могут быть предприняты и другие действия:

а)Тайм-аут в состоянии INITIAL: участник не может сообщить о своем решении координатору и не может зафиксировать транзакцию. Но может откатить свою часть транзакции. Если он позднее получит команду PREPARE, он может проигнорировать ее или отправить координатору сообщение ABORT.

б)Тайм-аут в состоянии PREPARED: участник уже известил координатор о решении COMMIT, то он не может его изменить. Участник оказывается заблокированным.

Действия, которые выполняются на отказавшем узле после его перезагрузки, называются ***протоколом восстановления***.

Они зависят от того, в каком состоянии находился узел, когда произошел сбой, и какую роль выполнял этот узел в момент отказа: координатора или участника.

▼ **Подробнее**

При отказе координатора:

- В состоянии INITIAL: процедура 2ФФ еще не запускалась, поэтому после перезагрузки следует ее запустить;

- в состоянии WAITING: координатор уже направил команду PREPARE, но еще не получил всех ответов и не получил ни одного сообщения ABORT. В этом случае он перезапускает процедуру 2ФФ;

- в состоянии DECIDED: координатор уже направил участникам глобальное решение. Если после перезапуска он получит все подтверждения, то транзакция считается успешно зафиксированной. В противном случае он должен прибегнуть к протоколу ликвидации.

При отказе участника цель протокола восстановления - гарантировать, что после восстановления узел выполнит в отношении транзакции то же действие, которое выполнили другие участники, и сделает это независимо от координатора, т.е. по возможности без дополнительных подтверждений.

Рассмотрим три возможных момента возникновения отказа:

- в состоянии INITIAL: участник еще не успел сообщит о своем решении координатору, поэтому он может выполнить откат, т.к. координатор не мог принять решение о глобальной фиксации транзакции без голоса этого участника;

- в состоянии PREPARED: участник уже направил сведения о своем решении координатору,

поэтому он должен запустить свой протокол ликвидации;

- в состоянии ABORTED/COMMITED: участник уже завершил обработку своей части транзакции, поэтому никаких дополнительных действий не требуется.

Обработка распределенных запросов и распределенная модель транзакций

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. 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:

- обработка запросов;

- управление каталогом;

- распространение обновления;

- управление восстановлением;

- управление параллелизмом.

Обработка запросов

Распределенным называется запрос, который обращается к двум и более узлам РБД, но не обновляет на них данные.

Запрашивающий узел должен определить, что в запросе идет обращение к данным на другом узле, выделить подзапрос к удаленному узлу и перенаправить его этому узлу.

Самой сложной проблемой выполнения распределенных запросов является **оптимизация**, т.е. поиск оптимального плана выполнения запроса. Информация, которая требуется для оптимизации запроса, распределена по узлам. Если выбрать центральный узел, который соберет эту информацию, построит оптимальный план и отправит его на выполнение, то теряется свойство локальной автономности.

Поэтому обычно распределенный запрос выполняется так: запрашивающий узел собирает все данные, полученные в результате выполнения подзапросов, у себя, и выполняет их соединение (или объединение), что может занять очень много времени.

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 *Ry*, содержащего сто кортежей на узле *Y*, и отношения *Rz*, содержащего миллион кортежей на узле *Z* .

Оптимизатор на узле *X* выберет глобальную стратегию для выполнения запроса *Q*, при этом, очевидно, лучше переместить отношение *Ry* на узел *Z*, а не отношение *Rz* на узел *Y* (и конечно же, не следует перемещать оба отношения *Ry* и *Rz* на узел *X*. Тогда сразу после перемещения отношения *Ry* на узел *Z* стратегия выполнения объединения на узле *Z* будет выбрана локальным оптимизатором на узле *Z*.

Пример: 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 = 100000

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 T[*i*] по следующей формуле:

T[*i*] = общая задержка доступа + (общий объем данных / интенсивность обмена данными) = (число сообщений / 10) +(число бит / 50000)

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

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

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

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

3 шаг. 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.

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

4 шаг. 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.

T[4] = приблизительно 2 с

5 шаг. 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 B.

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

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

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

Эти результаты представлены в таблице 19

Таблица 19 - Стратегии распределенного выполнения запроса (итоги)

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)

Внимательно ознакомившись с этими результатами, можно отметить следующие важные особенности:

- Каждая из шести стратегий представляет собой один из возможных подходов к решению этой проблемы, несмотря на очень значительные вариации во времени передачи данных.

- Интенсивность обмена данными и задержка доступа являются важными факторами, влияющими на выбор той или иной стратегии.

- Для плохих стратегий продолжительность операций вычисления и ввода-вывода данных пренебрежимо мала по сравнению со временем передачи данных.

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.

Централизованный каталог. Весь каталог хранится в одном месте, т.е. на центральном узле.

Полностью реплицированный каталог. Весь каталог полностью хранится на каждом узле.

Секционированный каталог. На каждом узле содержится его собственный каталог для объектов, хранимых на этом узле. Общий каталог является объединением всех разъединенных

локальных каталогов.

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 пользователи (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.

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*.

Приведем пример такой инструкции.

CREATE SYNONYM MSTATS FOR MARYLIN@NEWYORK.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*.

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:

- элемент каталога для каждого объекта, *созданного* на этом узле;

- 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.

Концепция репликации в системе с откладываемым распространением обновления может рассматриваться как ограниченное воплощение идеи снимков.

Одна из причин 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.

▼ **Подробнее**

- 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.

- При двухфазной фиксации требуется, чтобы координатор обменивался данными с каждым узлом-участником, что, в свою очередь, означает большее количество сообщений и больше накладных расходов.

- 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.

- 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 выполнение 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.

- 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. 4 представлен пример возникновения такого тупика:

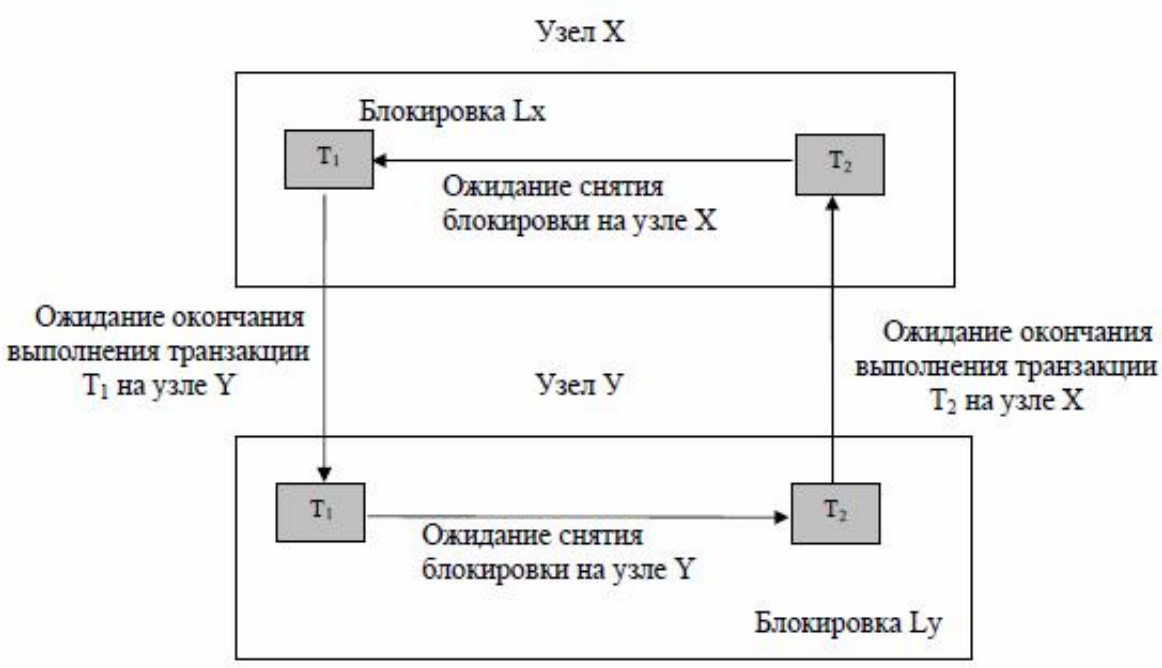

Рисунок 4 - Пример глобального тупика

▼ **Подробнее**

1. Агент транзакции T2 на узле X ожидает, когда агент транзакции T1 снимет блокировку на узле X

2. Агент транзакции T1 на узле X ожидает, когда закончится выполнение транзакции T1 на узле Y

3. Агент транзакции T1 на узле Y ожидает, когда агент транзакции T2 снимет блокировку на узле Y.

4. Агент транзакции T2 на узле Y ожидает, когда закончится выполнение транзакции

T2 на узле X. Тупиковая ситуация!

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.

Распределение и тиражирование данных

Одним из основных требований к распределенной базе данных остается требование наличия развитой методологии распределения и размещения данных, включая разбиение.

Технология COM

COM (Component Object Model) - это объектная модель компонентов. Данная технология является базовой для технологий ActiveX и OLE. Технологии OLE и ActiveX - всего лишь надстройки над данной технологией. В качестве примера можно привести объект TObject, как базовый объект VCL Delphi. Точно так же технология COM является базовой по отношению к OLE и ActiveX.

Технология COM применяется при описании API и двоичного стандарта для связи объектов различных языков и сред программирования. COM предоставляет модель взаимодействия между компонентами и приложениями.

Технология COM работает с так называемыми COM-объектами. COM-объекты похожи на обычные объекты визуальной библиотеки компонентов Delphi. В отличие от объектов VCL Delphi, COM-объекты содержат свойства, методы и интерфейсы.

Обычный COM-объект включает в себя один или несколько интерфейсов. Каждый из этих интерфейсов имеет собственный указатель.

Технология COM имеет два явных преимущества:

- создание COM-объектов не зависит от языка программирования. Таким образом, COM-объекты могут быть написаны на различных языках;

- COM-объекты могут быть использованы в любой среде программирования под Windows. В число этих сред входят Delphi, Visual C++, C++Builder, Visual Basic, и многие другие.

Хотя технология COM обладает явными плюсами, она имеет также и минусы, среди которых зависимость от платформы. То есть, данная технология применима только в операционной системе Windows и на платформе Intel.

Все COM-объекты обычно содержатся в файлах с расширением DLL или OCX. Один такой файл может содержать как одиночный COM-объект, так и несколько COM-объектов.

Ключевым аспектом технологии COM является возможность предоставления связи и взаимодействия между компонентами и приложениями, а также реализация клиент-серверных взаимодействий при помощи интерфейсов.

Технология COM реализуется с помощью *COM-библиотек* (в число которых входят такие файлы операционной системы, как OLE32.DLL и OLE-Aut32.DLL). COM-библиотеки содержат набор стандартных интерфейсов, которые обеспечивают функциональность COM-объекта, а также небольшой набор функций API, отвечающих за создание и управление COM-объектов.

В Delphi реализация и поддержка технологии COM называется *каркасом Delphi ActiveX* (Delphi ActiveX framework, DAX). Реализация DAX описана в модуле Axctris.

COM-объект

COM-объект представляет собой двоичный код, который выполняет какую-либо функцию и имеет один или более интерфейс.

COM-объект содержит методы, которые позволяют приложению пользоваться COM-объектом. Эти методы доступны благодаря COM-интерфейсам. Клиенту достаточно знать несколько базовых интерфейсов COM-объекта, чтобы получить полную информацию о составе свойств и методов объекта. COM-объект может содержать один или несколько интерфейсов. Для программиста COM-объект работает так же, как и класс в Object Pascal.

COM-интерфейсы

COM-интерфейс применяется для объединения методов COM-объекта. Интерфейс позволяет клиенту правильно обратиться к COM-объекту, а объекту - правильно ответить клиенту. Названия COM-интерфейсов начинаются с буквы I. Клиент может не знать, какие интерфейсы имеются у COM-объекта. Для того чтобы получить их список, клиент использует базовый интерфейс lunknown, который есть у каждого COM-объекта.

Пользователь COM-объекта

Пользователем COM-объекта называется приложение или часть приложения, которое использует COM-объект и его интерфейсы в своих собственных целях. Как правило, COM-объект находится в другом приложении.

COM-классы

COM со-классы (coclass) - это классы, которые содержат один или более COM-интерфейс. Вы можете не обращаться к COM-интерфейсу непосредственно, а получать доступ к COM-интерфейсу через со-класс. Со-классы идентифицируются при помощи идентификаторов класса (CLSID).

Библиотеки типов

COM-объекты часто используют библиотеки типов. *Библиотека типов* - это специальный файл, который содержит информацию о COM-объекте. Данная информация содержит список свойств, методов, интерфейсов, структур и других элементов, которые содержатся в COM-объекте. Библиотека типов содержит также информацию о типах данных каждого свойства и Типах данных, возвращаемых методами COM-объекта.

Файлы библиотеки типов имеют расширение TLB.

Технология DCOM

Технология DCOM (Distributed COM) - это распределенная COM-технология. Она применяется для предоставления средств доступа к COM-объектам, расположенным на других компьютерах в сети (в том числе и сети Internet).

Операционные системы Windows NT 4 и Windows 98 имеют встроенную поддержку DCOM.

Счетчики ссылок

Каждый COM-объект имеет счетчик ссылок. Данный счетчик содержит число процессов, которые в текущий момент времени используют COM-объект. Под процессом здесь подразумевается любое приложение или DLL, которые используют COM-объект, т. е. пользователи COM-объекта. Счетчик ссылок на COM-объект нужен для того, чтобы высвобождать процессорное время и оперативную *память, занимаемую COM-объектом, в том* случае, когда он не используется.

После создания и обращения к COM-объекту счетчик ссылок увеличивается на единицу. Всякий раз, когда новое приложение подключается к COM-объекту - счетчик увеличивается. Когда процесс отключается от COM-объекта - счетчик уменьшается. При достижении счетчиком нуля память, занимаемая COM-объектом, высвобождается.

OLE-объекты

Часть данных, использующаяся совместно несколькими приложениями, называется *OLE-объектом.* Те приложения, которые могут содержать в себе OLE-объекты, называются *OLE-контейнерами* (OLE container). Приложения, имеющие возможность содержать свои данные в OLE-контейнерах, называются *OLE-серверами* (OLE server).

Составные документы

Документ, включающий в себя один или несколько OLE-объектов, называется *составным документом.* Приложение, которое может содержаться внутри документа, называется *ActiveX-документом* (ActiveX document).

Остальные термины, присущие технологии COM, мы рассмотрим в следующих разделах данной книги.

Состав COM-приложения

При создании COM-приложения необходимо обеспечить следующее:

- COM-интерфейс;

- COM-сервер;

- COM-клиент.

Рассмотрим эти три составляющие COM-приложения более подробно.

COM-интерфейс

Клиенты COM связываются с объектами при помощи COM-интерфейсов. *Интерфейсы* -- это группы логически или семантически связанных процедур, которые обеспечивают связь между поставщиком услуги (сервером) и его клиентом. На рис. 3.1 схематично изображен стандартный COM-интерфейс.

Ключевыми аспектами COM-интерфейсов являются следующие:

- Однажды определенные, интерфейсы не могут быть изменены. Таким образом, вы можете возложить на один интерфейс определенный набор функций. Дополнительную функциональность можно реализовать с помощью дополнительных интерфейсов.

-По взаимному соглашению, все имена интерфейсов начинаются с буквы *I,* например IPersist, IMalloc.

-Каждый интерфейс гарантированно имеет свой уникальный идентификатор, который называется *глобальный уникальный идентификатор* (Globally Unique Identifier, GUID). Уникальные идентификаторы интерфейсов называют *идентификаторами интерфейсов* (Interface Identifiers, IIDs). Данные идентификаторы обеспечивают устранение конфликтов имен различных версий приложения или разных приложений.

- Интерфейсы не зависят от языка программирования. Вы можете воспользоваться любым языком программирования для реализации COM-интерфейса. Язык программирования должен поддерживать структуру указателей, а также иметь возможность вызова функции при помощи указателя явно или неявно.

- Интерфейсы не являются самостоятельными объектами, они лишь обеспечивают доступ к объектам. Таким образом, клиенты не могут напрямую обращаться к данным, доступ осуществляется при помощи указателей интерфейсов.

- Все интерфейсы всегда являются потомками базового интерфейса Iunknown.

Архитектура CORBA

CORBA (Common Object Request Broker Architecture) - обобщенная архитектура брокера объектных запросов, разработана Группой управления объектами (Object Management Group, OMG).

Строго говоря, если судить по ее названию, CORBA представляет собой не столько распределенную систему, сколько ее спецификацию

CORBA определяет механизм, обеспечивающий взаимодействие приложений в распределенной системе.

Главными компонентами стандарта CORBA являются:

- **объектный брокер запросов (Object Request Broker);**

- **язык определения интерфейсов (Interface Definition Language).**

В спецификацию CORBA включено также несколько дополнительных, но очень важных сервисов:

- **сервис динамического формирования запросов (DII);**

- **сервис репозитория интерфейсов (IR);**

- **сервис динамической обработки запросов (DSI);**

- **сервис, обеспечивающий взаимодействие различных брокеров запросов (GIOP).**

Концептуально спецификация CORBA относится к двум верхним уровням семиуровневой модели взаимодействия открытых систем. Характерные особенности проведения разработок в технологии CORBA заключаются в следующем:

- Язык описания интерфейсов OMG IDL позволяет определить интерфейс, независимый от языка реализации.

- Высокий уровень абстракции CORBA в семиуровневой модели OSI избавляет программиста от работы с низкоуровневыми сетевыми протоколами.

- Программисту не требуется информация о месте сервера в сетевой ИС и способе его активации.

- Разработка клиентской программы не зависит от серверной операционной системы и аппаратной платформы.

Объектная модель CORBA

Объектная модель CORBA определяет взаимодействие между клиентами и серверами.

Клиенты - это приложения, которые запрашивают сервисы, предоставляемые серверами.

Объекты-серверы содержат набор сервисов, разделяемых между многими клиентами.

Операция указывает запрашиваемый сервис.

Интерфейсы объектов описывают множество операций, которые могут быть вызваны клиентами определенного объекта.

Реализации объектов - это приложения, исполняющие сервисы, запрашиваемые клиентами.

Глобальная архитектура CORBA приведена на рис. 4.

Рисунок 4 - Архитектура CORBA

Объектный брокер запросов (ORB)

Спецификация CORBA разработана для обеспечения возможности интеграции разных объектных систем. На рисунке показано место главного компонента спецификации - брокера объектных запросов.

Задачей брокера является предоставление механизма выполнения запроса объекта-клиента: поиск объекта, к которому относится данный запрос, передача необходимых данных, подготовка объекта к обработке. Брокер объектных запросов обеспечивает прозрачное взаимодействие клиентского и серверного приложений.

Для разработчика вызов методов удаленных объектов не отличается от обычных локальных вызовов.

Естественно, обработка вызовов разных видов происходит различными способами. Вызов удаленного объекта обрабатывается особыми методами, определенными в CORBA-спецификации. Они формируют по сделанному запросу низкоуровневое представление, зависящее от используемых аппаратно-программных средств.

Клиент может запрашивать выполнение операций с помощью ORB несколькими способами.

Вызов операций разделяемого объекта-сервера может быть статическим, через IDL-суррогат, или динамическим (Dynamic Invocation Interface). В случае статического вызова описания интерфейсов на IDL отображаются в программный код на языках C, C++, Smalltalk и др. При использовании динамического интерфейса запросы формируются специальным образом, без отображения интерфейса объекта в исходный код разрабатываемого приложения.

Информация об интерфейсах объектов может быть получена клиентом во время компиляции или во время выполнения. Интерфейсы могут быть также указаны с помощью службы репозитория интерфейсов (Interface Repository). Этот сервис представляет интерфейсы как объекты, обеспечивая доступ к ним во время работы приложения.

Брокер объектных запросов является промежуточным слоем, обеспечивающим объединение информационных ресурсов распределенной неоднородной системы. В этом смысле бр**окер запросов есть основа интеграционной архитектуры**, необходимой для разработки распределенной системы масштаба корпорации.

Достоинства CORBA

- Язык IDL поддерживает разнообразные программные языки, операционные системы, сети и объектные системы. IDL позволяет отделить описание интерфейса от его реализации. Таким образом, можно менять объекты, не затрагивая интерфейсы. Приложение, даже если оно написано не на объектно-ориентированным языке, с помощью IDL можно инкапсулировать в объектную структуру.

- CORBA - сетевая архитектура по определению, эта идея лежит в основе его развития. Объектно-ориентированные интерфейсы CORBA легко определять, создавать и использовать.

- Каждый сервер может содержать много объектов. Связь между отправителем и адресатом осуществляется напрямую. Объекты могут быть разных размеров.

- CORBA хорошо сочетается с разнообразным промежуточным ПО, включая OLE языки (например, для реализации интерфейса можно использовать VisualBatch).

- В рамках CORBA можно обеспечить необходимый уровень безопасности системы.

- Интеграция с другими распространенными технологиями: базами данных, системами обработки сообщений, системами обработки пользовательского интерфейса и другими. Специализация по отраслям промышленности открывает дополнительные возможности для приближения объектов к реальным структурам.

- Существует протокол IIOP, который позволяет взаимодействовать различным ORB по TCP/IP. CORBA сервисы обеспечивают ряд дополнительных возможностей: транзакции, события, query и т. д. Одновременная поддержка статических и динамических интерфейсов. Возможность включения в распределенную среду Web-клиентов и серверов, в частности, через Java-реализации CORBA.

- CORBA - широко используемый стандарт, со множеством реализаций, но создается и поддерживается он централизованно, OMG.

- Так как CORBA - только стандарт, между его реализациями естественное возникает соревнование. Тем самым повышается качество продуктов, а их совместимость заложена в стандарте.

- По мере развития CORBA процесс создания программных приложений все больше напоминает конструирование из готовых деталей.

Секционирование таблиц

Секционирование делает большие таблицы и индексы более управляемыми, так как позволяет быстро и эффективно получать доступ к поднаборам данных и управлять ими, при этом сохраняя целостность всей коллекции. При использовании секционирования такие операции, как загрузка данных из системы OLTP в систему OLAP, занимают всего несколько секунд вместо минут и часов, затрачивавшихся на это в предыдущих версиях SQL Server. Операции обслуживания, выполняемые на поднаборах данных, также выполняются значительно эффективнее, так как нацелены только на те данные, которые действительно необходимы, а не на всю таблицу.

Данные секционированных таблиц и индексов подразделяются на блоки, которые могут быть распределены по нескольким файловым группам в базе данных. Данные секционируются горизонтально, поэтому группы строк сопоставляются с отдельными секциями. Таблица или индекс рассматриваются как единая логическая сущность при выполнении над данными запросов или обновлений. Все секции одного индекса или таблицы должны находиться в одной и той же базе данных.

Секционированные таблицы и индексы поддерживают все свойства и возможности, связываемые с разработкой и опрашиванием стандартных таблиц и индексов, включая ограничения, значения по умолчанию, значения идентификации и отметок времени, а также триггеры. Таким образом, если необходимо реализовать секционированное представление, локальное для данного сервера, вместо этого можно реализовать секционированную таблицу.

Решение, стоит ли применять секционирование, в основном зависит от того, насколько велика таблица или насколько она может увеличиться, как она используется и насколько эффективно отвечает на пользовательские запросы и операции обслуживания.

В целом, большую таблицу стоит секционировать, если выполняются следующие два условия.

- Таблица содержит (или может в будущем накопить) множество данных, используемых различными способами.

- Запросы или обновления таблицы выполняются не так, как ожидалось, либо затраты на обслуживание превышают прогнозируемые периоды технического обслуживания.

Например, если текущий месяц используется, в основном, для операций INSERT, UPDATE, DELETE и MERGE, в то время как предыдущие месяцы используются, в основном, для запросов SELECT, работа с этой таблицей может быть упрощена, если таблица секционирована на месяцы. Это особенно удобно, если операции регулярного обслуживания таблицы нацелены только на некоторый поднабор данных. Если таблица не секционирована, выполнение таких операций требует большого количества ресурсов и задействует весь набор данных. Когда применяется секционирование, такие операции обслуживания, как перестроение индекса и дефрагментация, можно выполнять, например только для одного месяца и данных, доступных только для записи, сохраняя доступ в сети к данным, доступным только для чтения.

Чтобы расширить этот пример, предположим, что нужно переместить данные, доступные только для чтения и относящиеся к одному месяцу, из этой таблицы в таблицу-хранилище данных для анализа. Благодаря секционированию поднаборы данных можно быстро разделить на отдельные участки обслуживания вне сети и добавлять их в виде секций к существующим секционированным таблицам, предполагая, что эти таблицы находятся в том же экземпляре базы данных. Такие операции обычно занимают всего несколько секунд вместо минут и часов, как это было в предыдущих выпусках.

Наконец, секционирование таблицы или индекса может улучшить производительность запросов, конечно, если оно правильно спроектировано с учетом типов часто выполняемых запросов и аппаратной конфигурации.

Фрагментация данных

База данных физически распределяется по узлам компьютерной информационной системы при помощи фрагментации и репликации (тиражирования) данных.

Отношения, принадлежащие реляционной базе данных, могут быть фрагментированы на горизонтальные или вертикальные разделы.

Горизонтальная фрагментация реализуется при помощи операции селекции, которая направляет каждый кортеж отношения в один из разделов, руководствуясь предикатом фрагментации. Например, для отношения Employee (Сотрудник) возможна фрагментация в соответствии с территориальным распределением рабочих мест сотрудников.

Тогда запрос "получить информацию о сотрудниках компании" может быть сформулирован так:

SELECT * FROM employee@donetsk,

employee@kiev

На рисунке 5 изображен принцип разделения данных при горизонтальной фрагментации.

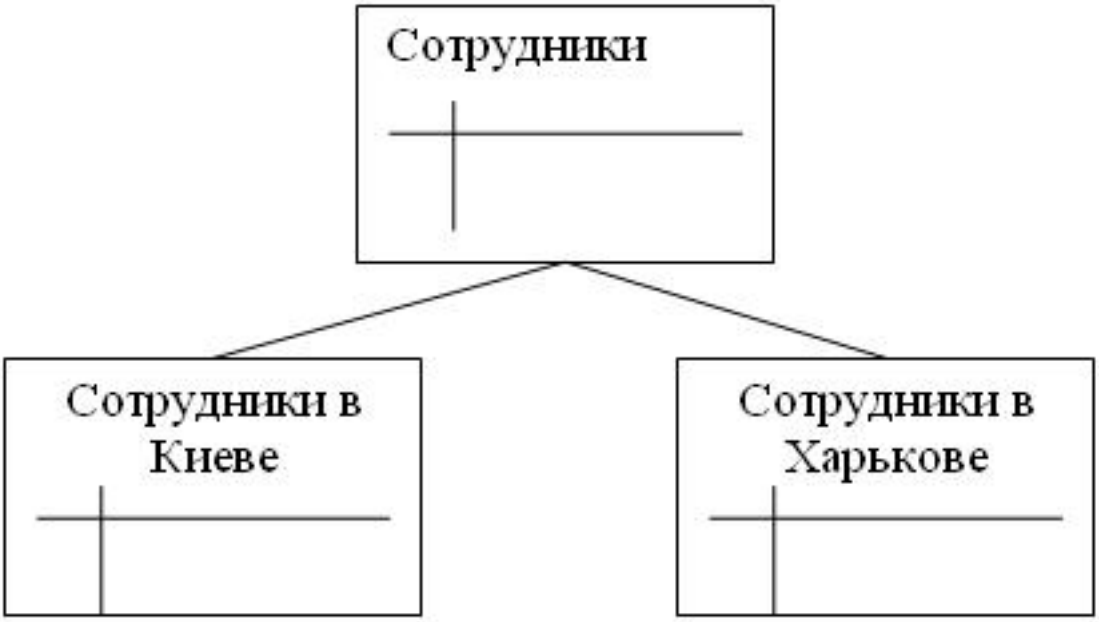

Рисунок 5 - Горизонтальная фрагментация

На рисунке 6 приведен пример горизонтальной фрагментации.

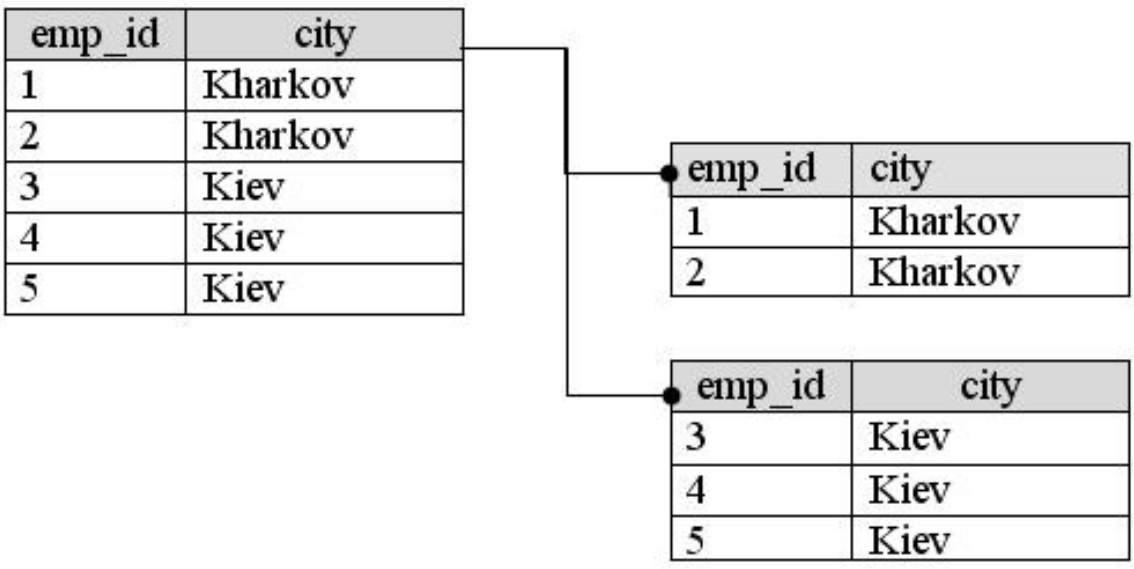

Рисунок 6 - Пример горизонтальной фрагментации

При вертикальной фрагментации отношение делится на разделы при помощи операции проекции. Например, один раздел отношения Employee может содержать поля Номер_сотрудника (emp_id), ФИО_сотрудника (emp_name), Адрес_сотрудника (emp_adress), а другой - поля Номер_сотрудника (emp_id), Оклад (salary), Руководитель (emp_chief).

Тогда запрос "получить информацию о заработной плате сотрудников компании" будет выглядеть следующим образом:

```
SELECT employee.emp_id,

emp_name,

salary

FROM employee@Kharkov,

employee@Kiev

ORDER BY emp_id
```

На рисунках 7 и 8 изображены сущность и пример вертикальной фрагментации.

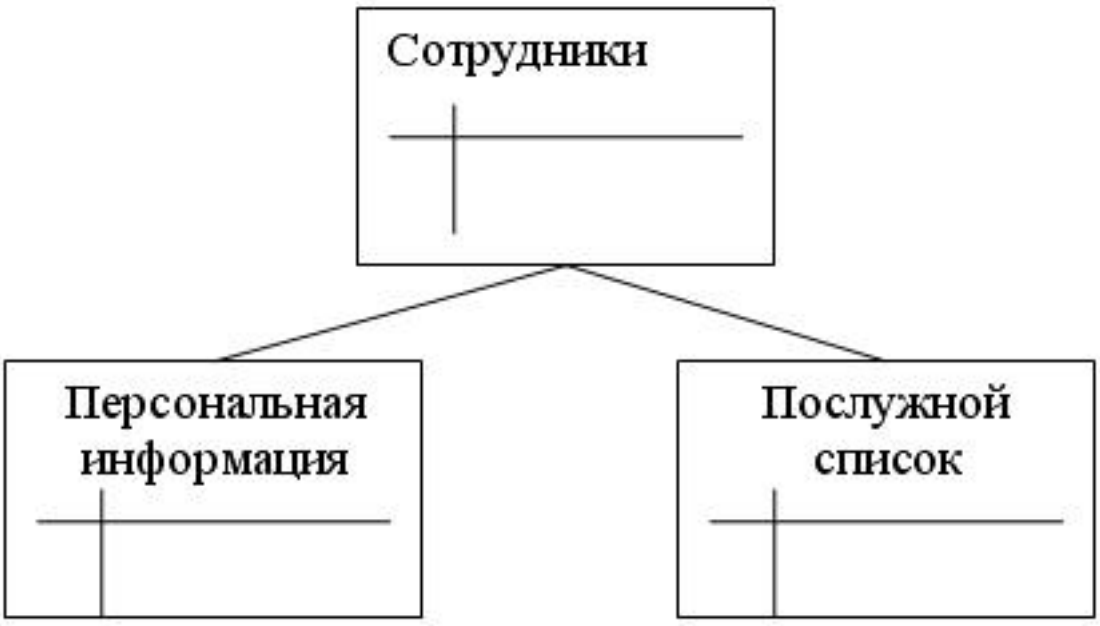

Рисунок 7 - Вертикальная фрагментация

Emp_id	Emp_name	Emp_address	Salary	Chief
1	Иванов Иван Иванович	Харьков, пр. Правды, 1, кв.5	1570	4
2	Белка Анна Ивановна	Харьков, ул. Артема, 9, кв.4	986	4
3	Фунт Игорь Игоревич	Киев, пер Вишневый, 1, кв.1	1255	4
4	Волк Игорь Игоревич	Киев, ул. Садовая, 101, кв.99	2552	1
5	Петров Петр Петрович	Киев, ул. Радужная, 1, кв. 10	1570	4

Emp_id	Emp_name	Emp_address
1	Иванов Иван Иванович	Харьков, пр. Правды, 1, кв.5
2	Белка Анна Ивановна	Харьков, ул. Артема, 9, кв.4
3	Фунт Игорь Игоревич	Киев, пер Вишневый, 1, кв.1
4	Волк Игорь Игоревич	Киев, ул. Садовая, 101, кв.99
5	Петров Петр Петрович	Киев, ул. Радужная, 1, кв. 10

Emp_id	Salary	Chief
1	1570	4
2	986	4
3	1255	4
4	2552	1
5	1570	4

Рисунок 8 - Пример вертикальной фрагментации

За счет фрагментации данные приближаются к месту их наиболее интенсивного использования, что потенциально снижает затраты на пересылки; уменьшаются также размеры отношений, участвующих в пользовательских запросах. Однако практически добиться ускорения выполнения запросов, затрагивающих фрагментированные отношения, очень трудно. Основная проблема состоит в резком расширении пространства поиска вариантов выполнения запросов, с которым должен работать оптимизатор запросов.

Репликация данных

Второй способ распределения данных - репликация (см. рис.9).

Репликация - механизм синхронизации содержимого нескольких копий объекта.

Репликация - это процесс, под которым понимается копирование данных из одного источника на множество других и наоборот.

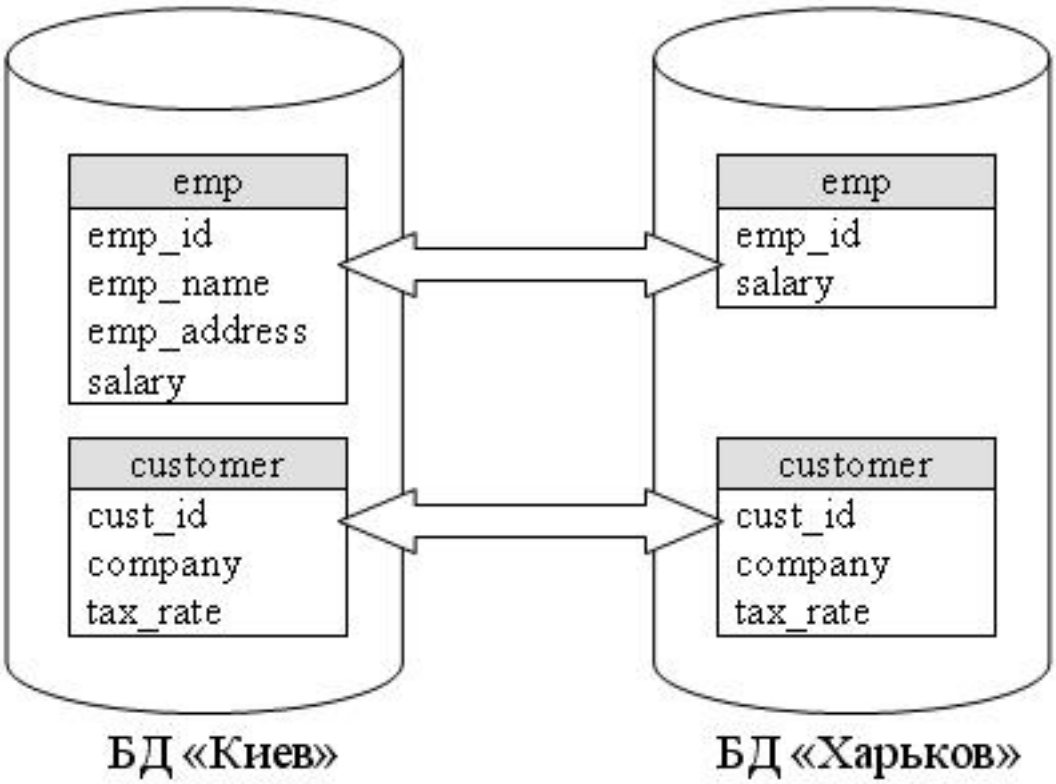

Рисунок 9 - Репликация

Репликаты - это множество различных физических копий некоторого объекта базы данных (обычно таблицы), для которых поддерживается синхронизация (идентичность) с некоторой "главной" копией.

При репликации изменения, сделанные в одной копии объекта, могут быть распространены в другие копии.

Репликация может быть синхронной или асинхронной.

Синхронная репликация

В случае синхронной репликации, если данная реплика обновляется, все другие реплики того же фрагмента данных также должны быть обновлены в одной и той же транзакции. Логически это означает, что существует лишь одна версия данных.

В большинстве продуктов синхронная репликация реализуется с помощью триггерных процедур (возможно, скрытых и управляемых системой). Но синхронная репликация имеет тот недостаток, что она создаёт дополнительную нагрузку при выполнении всех транзакций, в которых обновляются какие-либо реплики (кроме того, могут возникать проблемы, связанные с доступностью данных).

Асинхронная репликация

В случае асинхронной репликации обновление одной реплики распространяется на другие спустя некоторое время, а не в той же транзакции. Таким образом, при асинхронной репликации вводится задержка, или время ожидания, в течение которого отдельные реплики могут быть фактически неидентичными (то есть определение реплика оказывается не совсем подходящим, поскольку мы не имеем дело с точными и своевременно созданными копиями).

В большинстве продуктов асинхронная репликация реализуется посредством чтения журнала транзакций или постоянной очереди тех обновлений, которые подлежат распространению. Преимущество асинхронной репликации состоит в том, что дополнительные издержки репликации не связаны с транзакциями обновлений, которые могут иметь важное значение для функционирования всего предприятия и предъявлять высокие требования к производительности.

К недостаткам этой схемы относится то, что данные могут оказаться несовместимыми (то есть несовместимыми с точки зрения пользователя). Иными словами, избыточность может проявляться на логическом уровне, а это, строго говоря, означает, что термин контролируемая избыточность в таком случае не применим.

Модели тиражирования

Теоретически значения всех данных в тиражированных объектах должны автоматически и незамедлительно синхронизироваться друг с другом. (На практике это правило обычно несколько ослабляется.) В некоторых системах копии используются исключительно в режиме чтения и обновляются в соответствии с заданным расписанием. В других средах допускается модификация отдельных значений в копиях, и эти изменения распространяются в соответствии с процедурами планирования и координации. На рисунках 10, 11 показаны различные модели тиражирования.

При репликации фрагменты данных тиражируются с учетом спроса на доступ к ним. Это полезно, если доступ к одним и тем же данным нужен из приложений, выполняющихся на разных узлах. В таком случае, с точки зрения экономии затрат, более эффективно будет поддерживать копии данных на всех узлах, чем непрерывно пересылать данные между узлами.

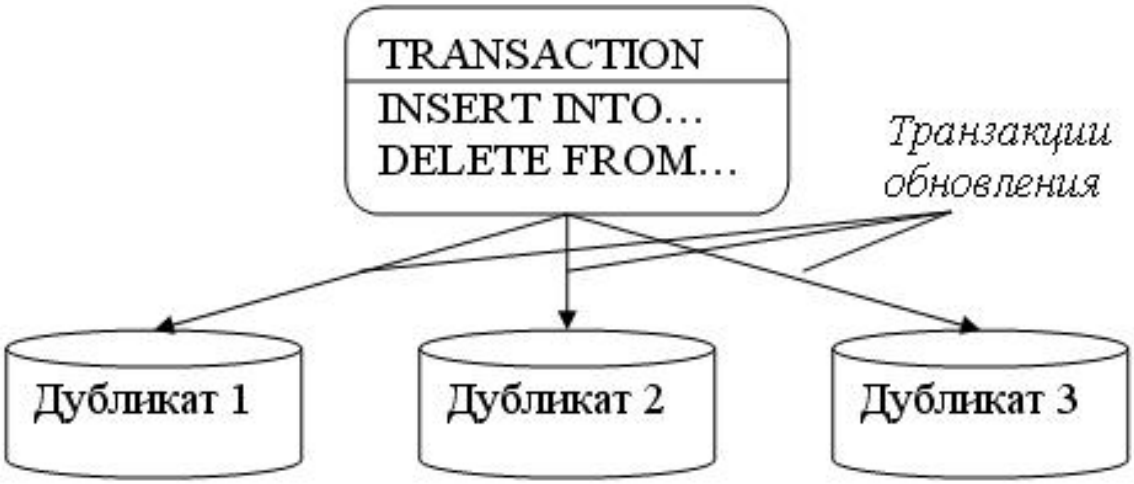

Рисунок 10 - Одновременное обновление (с управлением параллелизмом)

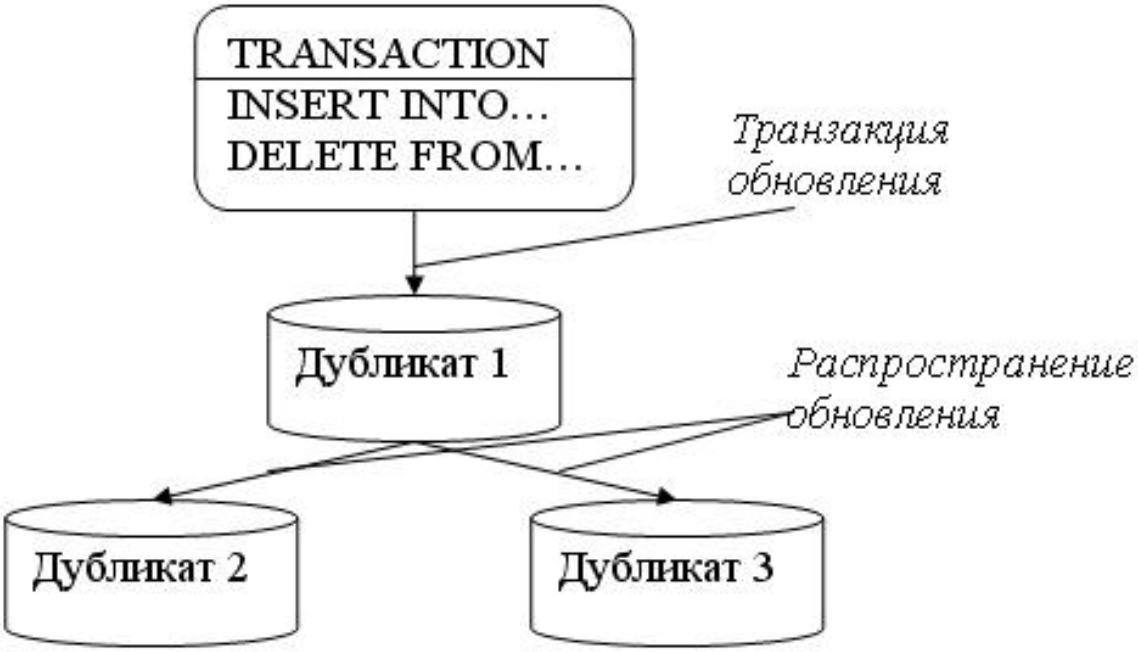

Рисунок 11 - Распространенные обновления

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 копий будет недоступна в текущий момент.

В современных СУБД функции репликации выполняет, как правило, специальный модуль - сервер тиражирования данных, называемый репликатором (так устроены СУБД - OpenIngress и Sybase). В Informix-OnLine Dynamic Server репликатор встроен в сервер, вOracle для использования репликации необходимо приобрести дополнительную опцию Replication Option.

Спецификация механизмов репликации зависит от используемой СУБД. Простейший вариант - использование "моментальных снимков" (snapshot).

Каталог распределенной системы

Важным компонентом структуры логического уровня РБД является сетевой каталог, который обеспечивает эффективное выполнение основных функций управления РБД и содержит всю информацию, необходимую для обеспечения независимости размещения, фрагментации и репликации. Существует несколько вариантов хранения системного каталога. Ниже перечислены некоторые из этих вариантов.

- Централизированный каталог. Весь каталог храниться в одном м месте, т.е. на центральном узле.

- Полностью реплицированный каталог. Весь каталог полностью хранится на каждом узле.

- Секционированный каталог. На каждом узле содержится его с собственный каталог для объектов, хранимых на этом узле. Общий каталог я является объединением всех разъединенных локальных каталогов.

- Комбинация первого и третьего вариантов. На каждом узле с содержится его собственный 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 распространять на каждый узел. В третьем выполнение не локальных 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".

Моментальные снимки в Oracle

Oracle поддерживает два типа тиражирования:

- базовое - копия обеспечивает доступ "только для чтения".

- усовершенствованное - приложения могут считывать и обновлять тиражируемые копии таблиц по всей системе (поддерживается специальными средствами СУБД - **Replication Option**).

Базовое тиражирование осуществляется (после установления связи с удаленной БД) с помощью создания моментальных снимков (snapshot), например:

CREATE SNAPSHOT sales.parts AS

SELECT * FROM sales.parts@central.compworld;

Моментальные снимки бывают:

- простые - создаются по однотабличному запросу SELECT, содержащему простые условия отбора.

- сложные - создаются по запросам, содержащим сложные условия отбора, фразы group by, having, обращающимся к двум и более таблицам и проч.

С помощью моментального снимка в локальной базе данных будет создано несколько объектов, поэтому пользователь, создающий моментальный снимок, должен иметь привилегии CREATE TABLE, CREATE VIEW и CREATE INDEX.

Синтаксис создания моментального снимка:

create snapshot [*имя_схемы*.]*имя_снимка*

[{ pctfree *целое* | pctused *целое* | initrans *целое* |

maxtrans *целое* | tablespace *имя_табличной_области* |

storage *размер_памяти* }]

[cluster *имя_кластера* (*имя_столбца*[,…])]

[using index]

[{ pctfree *целое* | pctused *целое* | initrans *целое* |

maxtrans *целое* | tablespace *имя_табличной_области* |

storage *размер_памяти* }]

[refresh [{ fast | complete | force }]

[start with *дата_1*] [next *дата_2*]]

[for update]

as *запрос*;

При создании моментального снимка в локальной базе данных создается:

- **таблица** для хранения записей, получаемых в результате выполнения запроса моментального снимка (с именем SNAP$_имя_моментального_снимка);

- **представление** этой таблицы "только для чтения", называемое в соответствии с именем моментального снимка;

- **представление**, называемое MVIEW$_имя_моментального_снимка - для обращения к удаленной основной таблице (или таблицам). Это представление будет использоваться во время регенерации.

Для модификации снимка, например, с целью установки частоты автоматического изменения в 1 час можно воспользоваться командой ALTER SNAPSHOT:

```
alter snapshot emp_dept_count refresh complete

start with sysdate next sysdate + 1/24;
```

Для удаления моментальных снимков применяется команда drop snapshot:

```
drop snapshot emp_dept_count;
```

Текущий контроль знаний

Транзакции и блокировки

Управление параллельным доступом

Что такое фиксатор?

- ❑ Блокировки, удерживаемые в течении непродолжительного времени (например для изменения структуры данных в памяти
- ❑ Блокировки, которые позволяют иметь доступ к данным только владельцу блокировки.
- ❑ Механизм SQL для управления параллельными операциями.

Управление параллельным доступом

Если пользователи только считывают данные и на их основе принимают определенные решения, меняющие данные, то когда становится заметным изменение данных для всех пользователей?

- ❑ До фиксации изменений.
- ❑ Сразу после внесения изменений одним из пользователей.
- ❑ После блокировки чтения пользователем, изменяющим данные.
- ❑ После выполнения фиксации изменений пользователем.

Управление параллельным доступом

Выберите верные утверждения

- ❑ Oracle блокирует данные на уровне строк.
- ❑ Oracle блокирует данные на уровне блоков и таблиц.
- ❑ Oracle блокирует данные с целью считывания.
- ❑ Сеанс записи блокируется, только если другой сеанс записи уже заблокировал строку.
- ❑ Сеанс считывания блокирует сеанс записи.
- ❑ Операции чтения не блокируется операциями записи.

Управление параллельным доступом

Что такое исключительные блокировки?

- ❑ Блокировки, удерживаемые в течении непродолжительного времени (например для изменения структуры данных в памяти
- ❑ Блокировки, которые позволяют иметь доступ к данным только владельцу блокировки.
- ❑ Механизм SQL для управления параллельными операциями.
- ❑ Блокировки, которые отслеживают возникновение конфликтов и при необходимости выполняют откат транзакций.

Управление параллельным доступом

Что такое блокировка?

- ❑ Объект базы данных, который позволяет управлять одновременным доступом к общему ресурсу.
- ❑ Механизм, используемый для реализации многопользовательской среды.

❑ Механизм, используемый для управления одновременным доступом к общему ресурсу.

Управление параллельным доступом

Установите уровни изоляциии в порядке уменьшения возможности возникновения аномалий баз данных

	REPEATABLE READ
	SERIALIZABLE
	READ COMMITED
	READ UNCOMMITED

Распределенные базы данных

Распределенные базы знаний

В чем заключается принцип аппаратной независимости для распределенных БД?

❑ Возможность запускать одну и ту же СУБД на различных аппаратных платформах

❑ Пользователи не должны знать, где именно данные хранятся физически и должны поступать так, как если бы все данные хранились на их собственном локальном узле

❑ Возможность функционирования СУБД под различными операционными системами

❑ Возможность поддерживать много принципиально различных узлов, отличающихся оборудованием и операционными системами, а также ряд типов различных коммуникационных сетей

Распределенные базы знаний

При какой архитектуре на сервере БД исполняется вся бизнес логика, которая реализована в виде хранимых процедур?

- ❑ Файл-сервер
- ❑ Клиент-сервер
- ❑ N-уровневая архитектура
- ❑ Выделенный сервер БД

Распределенные базы знаний

Что такое репликация данных?

- ❑ разделение хранимых объектов баз данных на отдельные части с раздельными параметрами физического хранения
- ❑ механизм синхронизации содержимого нескольких копий объекта
- ❑ разделение данных на уровне ресурсовавтоматическое разбиение элементов некоторого множества на группы в зависимости от их схожести

Распределенные базы знаний

Какая технология применяется при описании API и двоичного стандарта для связи объектов различных языков и сред программирования?

- ❑ Технология DCOM
- ❑ Технология COM
- ❑ OLE-объекты

- ❑ Object Request Broker
- ❑ CORBA

Распределенные базы знаний

Какой элемент программной системы обеспечивает прозрачное взаимодействие клиентского и серверного приложений?

- ❑ язык определения интерфейсов
- ❑ сервис динамического формирования запросов (DII)
- ❑ объектный брокер запросов
- ❑ сервис репозитория интерфейсов (IR)
- ❑ сервис динамической обработки запросов

Распределенные базы знаний

Какой способ хранения каталога не поддерживается в распределенных БД?

- ❑ Полностью реплицированный каталог
- ❑ Секционированный каталог
- ❑ Кластеризованный каталог
- ❑ Централизованный каталог

Словарь терминов