### Лекция 4. Принципы распределенной обработки данных

1. **Непрерывное функционирование**
2. **Обработка распределенных запросов.**
3. **Управление распределенными транзакциями**
4. **Автоматическое** **преобразование форматов данных;**
5. **Автоматическая** **трансляция кодов;**
6. **Межоперабельность (шлюзы)**

В основу взаимодействия прикладных программ — клиентов и сервера базы данных, положен ряд фундаментальных принципов, определяющих функциональные возможности современных СУБД в части, касающейся сетевого взаимодействия и распределенной обработки данных, среди которых:

1. **Непрерывное функционирование**

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

Надежность (вероятность того, что система исправна и работает в любой заданный момент) повышается благодаря работе распределенных систем не по принципу «все или ничего», а в постоянном режиме; т.е. работа системы продолжается, хотя и на более низком уровне, даже в случае неисправности некоторого отдельного компонента, например отдельного узла.

Доступность (вероятность того, что система исправна и работает в течении некоторого промежутка времени) повышается частично по той же причине, а частично благодаря возможности репликации данных.

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

**2.Обработка распределенных запросов.**

Рассмотрим два важных вопроса.

2.1. Преимущество **реляционной модели в** **производительности**

Рассмотрим запрос «Получить сведения о находящихся в Харькове поставщиках красных деталей», причем допустим, что пользователь находится в Киеве а данные хранятся на узле в Лондоне. Предположим, также, что такому запросу удовлетворяют n поставщиков. Если система является реляционной, то в запросе будут содержаться два сообщения – одно с запросом из Киева для Харькова, а другое с возвращаемым результатом, т.е. набором из n кортежей, которые пересылаются из Харькова в Киев. С другой стороны, если система не является реляционной, а работает в режиме последовательной обработки данных, то в запрос будет включен 2n сообщений – n из Киева в Харьков с запросом сведений о «следующем» поставщике, а также n из Харькова в Киев для возвращения сведений о «следующем» поставщике. Этот пример иллюстрирует, что **реляционная система по производительности может на несколько порядков превосходить нереляционную** (по крайней мере на уровне множеств).

**2.2 Оптимизация распределенных запросов**

Цель оптимизации запросов состоит в увеличении пропускной способности и улучшения времени отклика при заданных ограничениях на компьютерное оборудование и программное обеспечение.

Основная проблема выполнения запроса состоит в том что для выполнения некоторого охватывающего несколько узлов запроса существует много способов перемещения данных по сети. В этом случае чрезвычайно важно найти наиболее эффективную стратегию.

Оптимизация обработки запросов более важена для распределенной системы, нежели для централизованной системы. Основная причина заключается в том, что для выполнения охватывающего несколько узлов запроса существует довольно много способов перемещения данных по сети. В таком случае чрезвычайно важно найти наиболее эффективную стратегию. Например, запрос на объединение отношения Rx, хранимого на узле Х, и отношения Ry, хранимого на узле Y, мог бы быть выполнен с помощью :

* перемещения отношения Rx на узел Y,
* перемещения отношения Ry на узел Х
* перемещения этих двух отношений на третий узел Z
* (т.д.)

при этом время отклика может варьироваться от доли секунды до нескольких часов. Для того чтобы добиться приемлемого времени отклика необходимо понимать поведение оптимизатора. Например шестая версия Oracle включает эвристическую, основанную на системе правил оптимизацию, тогда как Oracle7 предлагает оба вида оптимизации: как основанную на системе правил, так и построенную на анализе стоимости.. В любом случае оптимизация требует анализа планов выполнения запросов. Для этого Oracle имеет несколько весьма полезных утилит: EXPLAIN PLAN, TKPROF. При выполнении распределенного запроса происходят следующие процессы:

Декомпозиция запроса: если запрос обращается к объектам, расположенным на нескольких компьютерах-серверах, то этот запрос разбивается на несколько SQL-предложений. Каждое предложение отсылается на соответствующий узел для выполнения.

Соединение: строки, возвращаемые SQL-предложениями, полученными в результате декомпозиции, собираются в одном пункте (узле сети), где осуществляется их соединение. Теми пунктами, где производится соединение, можно управлять (см. пример ниже).

Коррелированные подзапросы: вообще говоря, использование такие подзапросы крайне невыгодны, даже если речь идет не о распределенном запросе, поскольку такой подзапрос будет обрабатываться не один раз за все время выполнение исходного запроса, а скорей всего при получении каждой новой строке исходного запроса. Если подзапрос обращается к удаленной таблице, то для каждой полученной строки исходного запроса подзапрос посылается по сети на удаленную базу данных для соответствующей обработки.

Индексы: локальный узел создаст временные таблицы для данных, получаемых с удаленных узлов. Как и при обычных (не распределенных) запросах, индексы не создаются для временных таблиц. Даже если удаленная таблица имеет уникальный индекс, он все равно не используется в операциях соединения. EXPRESSION, GROUP BY, ORDER BY, DISTINCT вычисляются и выполняются на локальных узлах. В приведенном ниже примере 'LOCAL' относится к тому узлу, с которым приложение устанавливает связь, а 'REMOTE' -- к другому узлу (или другим узлам) сети. Рассмотрим следующую ситуацию. Таблица ORDER расположена на узле LOCAL, имеет уникальный индекс на order\_row и 100.000 строк. Таблица CUSTOMER расположена на узле REMOTE, имеет уникальный индекс на cust\_id и 1 млн строк. На узле LOCAL определена связь с базой данной (database link) с REMOTE и был создан синоним "CUSTOMER" для таблицы CUSTONER. Следующий запрос был выдан приложением, связанным с узлом LOCAL. select cust\_name, order\_num from order, customer where order.cust\_id = customer.cust\_id; Запрос был декомпозирован в:

SELECT CUST\_ID, CUST\_NAME FROM CUSTOMER;

<== executed at REMOTE.

SELECT CUST\_ID, ORDER\_NUM FROM ORDER;

<== executed at LOCAL.

В результате 1 млн строк таблицы CUSTOMER будет послано на узел LOCAL, и соответственно сетевой трафик возрастет. На узле LOCAL будет создана временная таблица для строк, получаемых с узла REMOTE. Поскольку обычные временные таблицы не индексируются, индекс таблицы клиента не будет использован. В качестве альтернативного варианта можно переслать таблицу ORDER на узел REMOTE, осуществить соединение на узле REMOTE и вернуть результат на узел LOCAL. Это можно осуществить путем создания LINK и VIEW на узле REMOTE.

CREATE DATABASE LINK LOCAL USING Connect-string;

CREATE VIEW CUST\_ORDER AS SELECT ORDER\_NUM, CUST\_NAME FROM ORDER@LOCAL O, CUSTOMER C WHERE O. CUST\_ID = C.CUST\_ID;

Теперь может быть выполнено следующее предложение выбора:

SELECT CUST\_NAME, ORDER\_NUM FROM CUST\_ORDER@REMOTE;

В данном примере следует принимать во внимание не только то, какая таблица или сколько строк будут пересылаться по сети, но и затраты на выполнение соединения и число строк, возвращаемых по выполнении запроса.

Технологические советы

* Избегайте соединений больших таблиц с разных узлов.
* На удаленных узлах по возможности используйте 'where clauses'.
* Анализируйте и оптимизируйте декомпозицию запросов.
* По возможности используйте триггеры баз данных и хранимые процедуры.
* Используйте представления удаленных таблиц для оптимизации функций GROUP BY, DISTINCT. Избегайте кореллированных подзапросов, которые обращаются к объектам на нескольких узлах. Рассмотрите применение репликации данных, где это только возможно и удобно. Эта ситуация подробно рассматривается ниже, а здесь следует отметить, что на основе некоторых правдоподобных предложений можно составить шесть различных стратегий обработки такого запроса.

**3. Управление распределенными транзакциями**

Для нейтрализации негативных последствий физических и логических ошибок при обработке данных в базе данных используется механизм транзакций. Транзакция – логически связанная группа обновлений базы данных которая должна быть либо полностью выполненной либо не выполненной вовсе .

По другому определению под транзакцией понимают механизм который обеспечивает указанное требование Система программных компонент обеспечиваючих поддержку управления транзакциями называют диспетчером транзакций.

Различные среди программирования обеспечивают собственные средства управления транзакциями. Например в С# для этого используются методы класса DBConnection: StartTransaction для начала транзакции , Commit для подтверждения и Rollback для отката

Более общий механизм предусмотрен стандартом язика SQL

* SetTransaction - начать транзакцию,
* Commit – подтвердить
* Rollback-откатить.

Для разных СУБД имеются отличия в детальном синтаксисе команд, но суть сохраняется.

Существует два основных аспекта управления обработкой транзакций, а именно: управление восстановлением и управление параллелизмом, каждому из которых в распределенных системах должно уделяться повышенное внимание.

В распределенной системе выполнение одной транкзации может быть связано с исполнением кода на нескольких узлах, в частности она может выполнять операции обновления на нескольких узлах. В таком случае говорят, что каждая транкзация состоит из нескольких **агентов**, т.е. процессов, выполняемых на данном узле по указанию заданной транкзации. Системе необходимо указать, что некоторые два агента представляют собой части одной и той же транкзации, например, для того, что-бы избежать возникновения тупиковой ситуации для двух агентов, которые содержит одна и та же транкзация.

Управлении восстановлением. Протокол двухфазной фиксации распределенных транзакций.

Для подтверждения того, что в распределенном окружении дання транкзация являеться атомарной (типа все-или-ничего), система должна убедиться, что агенты заданного набора должны быть все вместе либо завершены, либо отменены. Эта цель может быть достигнута только за счет реализации протокола двухфазной фиксации, суть которого состоит в том что выполнение транзакции в распределенной базе данных управляет системный компонент называемый координатором под управлением которого работают независимые администраторы ресурсов узлов сети. После выполнения запроса на commit координатор осуществляет следующий двухфазный процесс:

* Каждый администратор ресурсов сохраняет все записи журнала регистрации вне собственных физических файлов ( вне енергозависимой памяти) регистрации для своих локальных ресурсов . Теперь, чтобы ни случилось администратор ресурса не будет выполнять постоянной записи от имени транзакции, а может при необходимости передавать все свои обновления и отменять их. Если насильственная запись прошла успешно, то администратор отвечает координатору ”ОК”, иначе = ” Not ОК”
* Когда координатор получил ответы от всех администраторов он насильственно заносит записи в свой физический файл регистрации указывая свое решение относительно всей транзакции. Если все ответы были ”ОК”, то решение будет „принять изменения” а если был хоть один ответ ”Not ОК” то ”отвергнуть изменения ”. Затем координатор информирует каждого администратора о своем решении, и каждый администратор согласно принятого решения локально фиксирует или аннулирует транзакцию

# Управление параллелизмом

Для управления параллелизмом, то в распределенных системах, так же как и в нераспределенных, используется метод блокировки. (В нескольких более современных системах уже используется средства многовариантного управления, однако в большинстве систем для этого все же продолжает использоваться метод блокировки).

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 Т, которая нуждается в обновлении объекта, имеющего реплики на п 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 5п сообщений:

п запросов на блокировку,

п разрешений на блокировку,

п сообщений об обновлении,

п подтверждений,

п 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 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 5п до 2п+3 (один запрос на блокировку, одно разрешение на блокировку, п обновлений, п подтверждений и один запрос на снятие блокировки). Однако обратите внимание, что 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 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 операции обновления.)

Другой проблемой блокировки в распределенной системе является то, что она может привести к глобальному тупику, который охватывает два или более узлов. На рисунке представлен пример возникновения такого тупика:

Каждая из транзакций выполняется на каждом из узлов , при этом ее выполнение происходит в три этапа: блокировка-выполнение - разблокировка

Транзакция Т1 стартовала на Х заблокировала узел Х и выполняется , но для завершения ей нужно ждать завершение Т1 на узле У

Транзакция Т1 стартовала на узле У, но тот заблокирован Т2

Транзакция Т2 стартовала на узле У заблокировала узел У и выполняется , но для завершения ей нужно ждать завершения Т2 на узле Х;

Транзакция Т2 стартовала на узле Х, но тот заблокирован Т1

Узел Х

Блокировка Lx

Т1

T2

Ожидание снятия блокировки на узле Х транкзакцией Т1

Ожидание окончания

выполнения транзакции Т1 на узле Y

Ожидание окончания выполнения транзакции Т2 на узле Х

Т1

Блокировка Ly

Т2

Ожидание снятия блокировки на узлеY транкзакцией Т2

Узел У

Агент транзакции Т1 на узле Х ожидает, когда закончится выполнение транзакции Т1 на узле У.

Агент транзакции Т1 на узле У ожидает, когда агент транзакции Т2 снимет блокировку на узле У.

Агент транзакции Т2 на узле У ожидает, когда закончится выполнение транзакции Т2 на узле Х.

Агент транзакции Т2 на узле Х ожидает, когда агент транзакции Т1 снимет блокировку на узле Х.

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 диаграм в глобальную диаграмму ожидания. Отсюда следует, что обнаружение глобальных тупиков связано с увеличением накладных расходов, поскольку для этого требуется дополнительно совместить отдельные локальные диаграммы.

**4.Автоматическое преобразование форматов данных**

Как только несколько компьютеров различных моделей под управлением различных операционных систем соединяются в сеть, сразу возникает вопрос о согласовании форматов представления данных. Действительно, в сети могут быть компьютеры, отличающиеся разрядностью (16-ти, 32-х и 64-х разрядные процессоры), порядком следования байт в слове, представлением чисел с плавающей точкой и т.д. Задача коммуникационного сервера состоит в том, чтобы на уровне обмена данными обеспечить согласование форматов между удаленным и локальным узлами с тем, чтобы данные, извлеченные сервером из базы на удаленном узле и переданные по сети, были правильно истолкованы прикладной программой на локальном узле.

**5.Автоматическая трансляция кодов**

В неоднородной компьютерной среде при взаимодействии клиента и сервера возникает также задача трансляции кодов. Сервер может работать с одной кодовой таблицей (например, EBCDIC), клиент — с другой (например, ASCII), при этом происходит рассогласование трактовки кодов символов. Поэтому, если на локальном узле используется одна кодовая таблица, а на удаленном — другая, то при передаче запросов по сети и при получении ответов на них необходимо обеспечить трансляцию кодов. Решение этой задачи также ложится на коммуникационный сервер. Мы рассмотрели детали взаимодействия одной пары "клиент-сервер". Однако в реальной жизни сервер базы данных должен обслуживать одновременно множество запросов от клиентов — следовательно, в один момент времени таких пар может быть несколько. И все проблемы взаимодействия, о которых речь шла выше, должны решаться коммуникационным сервером для всех этих взаимодействующих пар. Системы с архитектурой "один-к-одному" (см. Разд. Сервер базы данных) для обслуживания сервером базы данных одновременно множества клиентов вынуждены загружать отдельный коммуникационный сервер для каждой пары "клиент-сервер". В результате нагрузка на операционную систему увеличивается, резко возрастает общее число ее процессов, расходующих вычислительные ресурсы. Это — один из недостатков архитектуры "один-к-одному". Именно поэтому для современных распределенных СУБД важно иметь многопотоковый коммуникационный сервер, который берет на себя задачи сетевой поддержки множества клиентов, одновременно обращающихся к серверу. На каждом узле сети он поддерживает множество пар соединений "клиент-сервер" и позволяет существовать одновременно множеству независимых сеансов работы с базами данных.

1. **Межоперабельность (шлюзы)**

Для обеспечения независимости от СУБД необходимо чтобы СУБД на разных узлах поддерживали одинаковый интерфейс. Рассмотрим пример при котором на узле Х из узлов функционирует СУБД INGRESS, а на узле У СУБД Oracle, хотя в общем случае можно рассматривать сочетания различных СУБД.

Допустим что некоторый пользователь U на узле Х желает просмотреть данные из БД на узле Х и из базы данных на узле У. По определению он является пользователем INGRESS, а потому работа С РБД для него длжна походить на работу с БД INGRESS. Указанная проблема решается следующим образом: система INGRESS должна иметь специальную надстроечную программу которая называется **шлюзом** и взаимодействует с системой Oracle так, чтобы создавалось впечатление что ″работа Oracle выглядит также как и работа INGRESS″.

Такой шлюз должен выполнять следующие функции:

* Реализацию протокола обмена информацией между системами INGRESS и Oracle учетом форматов данных используемых в разных системах ;
* Должен обрабатывать произвольные незапланированные утверждения языка SQL для Oracle , то есть должен содержать динамическую поддержку языка SQL ;
* Взаимное отображение типов данных между системами INGRESS и Oracle , например кошмарное разнообразие способов хранения даты и времени;
* Отображение диалекта языка SQL для INGRESS на диалект языка SQL Oracle ;
* Отображение информации пересылаемой обратно от системы Oracle к системе INGRESS ;
* Отображение каталога Oracle в соответствующую структуру пользователя INGRESS, чтобы тому была понятна структура БД Oracle ;
* Поддерживать протоколы двухфазной фиксации транзакций для обеих систем;
* Поддерживать необходимые блокировки данных.