**Лекція 4. Принципи розподіленої обробки даних**

**1. Безперервне функціонування.**

**2. Обробка розподілених запитів.**

**3. Керування розподіленими транзакціями.**

**4. Автоматичне перетворення форматів даних.**

**5. Автоматична трансляція кодів.**

**6. Межоперабельність (шлюзи).**

В основу взаємодії прикладних програм - клієнтів і сервера бази даних, покладено ряд фундаментальних принципів, що визначають функціональні можливості сучасних СКБД у тій частині, яка стосується мережної взаємодії і розподіленої обробки даних, серед яких:

**1. Безперервне функціонування**

Однією з основних переваг розподілених систем є те, що вони забезпечують більш високу надійність і доступність.

***Надійність*** (імовірність того, що система справна і працює в будь-який заданий момент) підвищується завдяки роботі розподілених систем не за принципом "усі чи нічого", а у постійному режимі, тобто робота системи продовжується, хоча і на більш низькому рівні, навіть у випадку несправності деякого окремого компонента, наприклад, окремого вузла.

***Доступність*** (імовірність того, що система справна і працює протягом деякого проміжку часу) підвищується частково по тій же причині, а частково завдяки можливості реплікації даних.

Ці міркування можна застосовувати також для випадку незапланованого вимикання деякого компонента усередині системи, наприклад унаслідок якої-небудь несправності. Незаплановані вимикання вкрай небажані, але їхнього виникнення важко уникнути. В ідеальному випадку варто було б зовсім виключити заплановані вимикання, тобто не виключати систему при виконанні будь-яких операцій, наприклад при додаванні нового вузла чи відновленні СКБД на деякому вузлі.

**2.Обробка розподілених запитів**

Розглянемо два важливих питання.

**2.1. Перевага реляційної моделі в продуктивності**

Розглянемо запит "Одержати відомості про постачальників червоних деталей, що знаходяться в Харкові", причому припустимо, що користувач знаходиться в Києві, а дані зберігаються на вузлі в Лондоні. Припустимо, також, що такому запиту задовольняють n постачальників. Якщо система є реляційною, то в запиті будуть міститися два повідомлення - одне з запитом з Києва для Харкова, а інше з результатом, що повертається, тобто набором з n кортежів, що пересилаються з Харкова в Київ. З іншого боку, якщо система не є реляційною, а працює в режимі послідовної обробки даних, то в запит буде включено 2n повідомлень - n з Києва в Харків із запитом відомостей про "наступного" постачальника, а також n з Харкова в Київ для повернення відомостей про "наступного" постачальника. Цей приклад ілюструє, що *реляційна система по продуктивності може на кілька порядків перевершувати нереляційну* (принаймні на рівні множин).

**2.2 Оптимізація розподілених запитів**

Мета оптимізації запитів полягає у збільшенні пропускної здатності і поліпшенні часу відгуку при заданих обмеженнях на комп'ютерне устаткування і програмне забезпечення.

Основна проблема виконання запиту полягає в тому, що для виконання деякого запиту, який охоплює кілька вузлів, існує багато способів переміщення даних по мережі. У цьому випадку надзвичайно важливо знайти найбільш ефективну стратегію.

Оптимізація обробки запитів більш важлива для розподіленої системи, ніж для централізованої системи. Основна причина полягає в тому, що для виконання запиту, який охоплює кілька вузлів, існує досить багато способів переміщення даних по мережі. У такому випадку надзвичайно важливо знайти найбільш ефективну стратегію. Наприклад, запит на об'єднання відношення Rx, що зберігається на вузлі Х, і відношення Ry, що зберігається на вузлі Y, міг би бути виконаний за допомогою :

* переміщення відношень Rx на вузол Y;
* переміщення відношень Ry на вузол Х;
* переміщення цих двох відносин на третій вузол Z;
* (т.д.)

при цьому час відгуку може варіюватися від частки секунди до декількох годин. Для того щоб досягти прийнятного часу відгуку необхідно розуміти поведінку оптимізатора. Наприклад, шоста версія Oracle включає евристичну, засновану на системі правил оптимізацію, тоді як Oracle7 пропонує обидва види оптимізації: як засновану на системі правил, так і побудовану на аналізі вартості. У будь-якому випадку оптимізація вимагає аналізу планів виконання запитів. Для цього Oracle має кілька дуже корисних утиліт: EXPLAІ PLAN, TKPROF. При виконанні розподіленого запиту відбуваються наступні процеси:

Декомпозиція запиту: якщо запит звертається до об'єктів, розташованих на кількох комп'ютерах-серверах, то цей запит розбивається на кілька SQL-пропозицій. Кожна пропозиція відсилається на відповідний вузол для виконання.

Сполучення: рядки, що повертаються SQL-пропозиціями, отриманими в результаті декомпозиції, збираються в одному пункті (вузлі мережі), де здійснюється їх сполучення. Тими пунктами, де виконується сполучення, можна керувати (див. приклад нижче).

Корреляційні підзапити: взагалі, використання таких підзапитів вкрай невигідно, навіть якщо мова йде не про розподілений запит, оскільки такий підзапит буде оброблятися не один раз за увесь час виконання вихідного запиту, а скоріше усього при одержанні кожного нового рядка вихідного запиту. Якщо підзапит звертається до віддаленої таблиці, то для кожного отриманого рядка вихідного запиту підзапит посилається по мережі на віддаленну базу даних для відповідної обробки.

Індекси: локальний вузол створює тимчасові таблиці для даних, отиманих з віддалених вузлів. Як і при звичайних (не розподілених) запитах, індекси не створюються для тимчасових таблиць. Навіть якщо віддалена таблиця має унікальний індекс, він всерівно не використовується в операціях сполучення. EXPRESSІON, GROUP BY, ORDER BY, DІSTІNCT обчислюються і виконуються на локальних вузлах. У приведеному нижче прикладі 'LOCAL' відноситься до того вузла, з яким додаток установлює зв'язок, а 'REMOTE' - до іншого вузла (чи інших вузлів) мережі. Розглянемо наступну ситуацію. Таблиця ORDER розташована на вузлі LOCAL, має унікальний індекс на order\_row і 100.000 рядків. Таблиця CUSTOMER розташована на вузлі REMOTE, має унікальний індекс на cust\_іd і 1 млн рядків. На вузлі LOCAL визначений зв'язок з базою даних (database lіnk) з REMOTE і був створений синонім "CUSTOMER" для таблиці CUSTONER. Наступний запит був виданий додатком, зв'язаним з вузлом LOCAL.

select cust\_name, order\_num from order, customer where order.cust\_іd = customer.cust\_іd;

Запит був декомпозований у:

SELECT CUST\_ІD, CUST\_NAME FROM CUSTOMER;

<== executed at REMOTE.

SELECT CUST\_ІD, ORDER\_NUM FROM ORDER;

<== executed at LOCAL.

У результаті 1 млн рядків таблиці CUSTOMER буде послано на вузол LOCAL, і відповідно мережний трафік зросте. На вузлі LOCAL буде створена тимчасова таблиця для рядків, отриманих з вузла REMOTE. Оскільки звичайні тимчасові таблиці не індексуються, індекс таблиці клієнта не буде використаний. Як альтернативний варіант можна переслати таблицю ORDER на вузол REMOTE, здійснити з'єднання на вузлі REMOTE і повернути результат на вузол LOCAL. Це можна здійснити шляхом створення LІNK і VІEW на вузлі REMOTE.

CREATE DATABASE LІNK LOCAL USІNG Connect-strіng;

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

Тепер може бути виконана наступна пропозиція вибору:

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

У даному прикладі варто брати до уваги не тільки те, яка таблиця чи скільки рядків будуть пересилатися по мережі, але і витрати на виконання з'єднання і число рядків, що повертаються після виконання запиту.

Технологічні поради:

* Уникайте з'єднань великих таблиць з різних вузлів.
* На віддалених вузлах по можливості використовуйте 'where clauses'.
* Аналізуйте й оптимізуйте декомпозицію запитів.
* По можливості використовуйте тригери баз даних і збережені процедури.
* Використовуйте представлення віддалених таблиць для оптимізації функцій GROUP BY, DІSTІNCT.
* Уникайте кореляційних підзапитів, які звертаються до об'єктів на декількох вузлах.
* Розгляньте застосування реплікації даних, де це тільки можливо і зручно. Ця ситуація докладно розглядається нижче, а тут слід зазначити, що на основі деяких правдоподібних пропозицій можна скласти шість різних стратегій обробки такого запиту.

**3. Керування розподіленими транзакціями**

Існує два основних аспекти керування обробкою транзакцій, а саме: керування відновленням і керування паралелізмом, кожному з яких у розподілених системах повинна приділятися підвищена увага.

У розподіленій системі виконання однієї транкзації може бути зв'язане з виконанням коду на декількох вузлах, зокрема вона може виконувати операції відновлення на декількох вузлах. У такому випадку говорять, що кожна транкзація складається з декількох ***агентів***, тобто процесів, що виконуються на даному вузлі за вказівкою заданої транкзації. Системі необхідно вказати, що деякі два агенти являють собою частини однієї і тієї ж транкзації, наприклад, для того, щоб уникнути виникнення тупикової ситуації для двох агентів, які містить одна і та ж транкзація.

#### Керування відновленням. Протокол двофазової фіксації розподілених транзакцій

Для нейтралізації негативних наслідків від фізичних і логічний помилок при обробці даних в базі даних використовується механізм транзакцій. Транзакція - логічна одиниця роботи в рамках якої група оновлень повинна бути або повністю виконаної або не виконаної взагалі.

За іншим визначенням під транзакцією розуміється механізм забезпечує виконання вказаного умови. Система програмних компонент забезпечують підтримку транзакцій називають диспетчером транзакції.

Різні серед програмування надам свої засоби для керування транзакціямі. Наприклад в С# для цього використовуються методи класу DBConnection: StartTransaction для початку транзакції, Commit для підтвердження та Rollback для скасування.

Більш загальний механізм передбачений стандартом мови SQL

* SetTransaction - почати транзакцію,
* Commit - підтвердити
* Rollback-скасувати ("відкотити") транзакцію.

У розподілених системах з розподіленою базою даних механизм управління транзакціями має свої особливості. Диспетчер розподіленої транзакції називається (і є) глобальним диспетчером, а механізм підтримки транзакції на окремих вузлах називають агентом транзакції.

Для підтвердження того, що в розподіленому оточенні данна транкзація являється атомарною (типу усе-чи-нічого), система повинна переконатися, що агенти заданого набору повинні бути або всі разом довершені, або скасовані. Ця мета може бути досягнута тільки за рахунок реалізації протоколу двофазної фіксації, суть якого полягає в тому що виконанням транзакції в розподіленій базі даних керує системний компонент, який називається координатором, під керуванням якого працюють незалежні адміністратори ресурсів вузлів мережі. Після виконання запиту на commіt координатор здійснює наступний двофазний процес:

- Кожен адміністратор ресурсів зберігає всі записи журналу реєстрації поза власними фізичними файлами (поза енергозалежною пам'яттю) реєстрації для своїх локальних ресурсів. Тепер, що б не сталося адміністратор ресурсу не буде виконувати постійного запису від імені транзакції, а може при необхідності передавати усі свої відновлення і скасовувати їх. Якщо насильницький запис пройшов успішно, то адміністратор відповідає координатору "ОК", інакше = " Not ОК"

Коли координатор одержав відповіді від всіх адміністраторів він насильно заносить записи у свій фізичний файл реєстрації, вказуючи своє рішення щодо всієї транзакції. Якщо усі відповіді були "ОК", то рішення буде "прийняти зміни" а якщо була хоч одна відповідь "Not ОК" то "відкинути зміни". Потім координатор інформує кожного адміністратора про своє рішення, і кожен адміністратор згідно прийнятого рішення локально фіксує чи анулює транзакцію.

Розглянемо цей процес більш детально.

Координатор виконує протокол за наступним алгоритмом:

I. Фаза 1 (голосування).

1. Координатор заносить запис begin\_commit в системний журнал і забезпечує її перенесення з буфера в ОП на ВЗУ.

2. Відправляє всім учасникам команду PREPARE.

3. Чекає відповідей усіх учасників в межах встановленого тайм-ауту.

II. Фаза 2 (прийняття рішення).

4. Якщо учасник повертає повідомлення ABORT, координатор поміщає в системний журнал запис abort і забезпечує її перенесення з буфера в ОП на ВЗП; відправляє всім учасникам повідомлення GLOBAL\_ABORT і очікує відповідей усіх учасників в межах встановленого тайм-ауту.

5. Якщо учасник не відповідає протягом встановленого тайм-ауту, координатор вважає, що даний учасник відкотить свою частину транзакції і надходить відповідно.

6. Якщо учасник повертає повідомлення READY\_COMMIT, координатор оновлює список учасників, що надіслали відповіді.

7. Якщо позитивні відповіді надіслали всі учасники, координатор поміщає в системний журнал запис commit і забезпечує її перенесення з буфера в ОП на ВЗУ. Відправляє всім учасникам повідомлення GLOBAL\_COMMIT і очікує відповідей усіх учасників в межах встановленого тайм-ауту.

8. Після надходження підтверджень про фіксацію від усіх учасників поміщає в системний журнал запис end\_transaction і забезпечує її перенесення з буфера в ОП на ВЗУ.

9. Якщо деякі вузли не прислали підтвердження фіксації, координатор заново направляє їм повідомлення про прийняте рішення і надходить за цією схемою до отримання всіх необхідних підтверджень.

Учасник транзакції діє за таким алгоритмом:

1. При отриманні команди PREPARE, в тому випадку, якщо він готовий зафіксувати свою частину транзакції, він поміщає запис ready\_commit в файл журналу транзакцій і відправляє координатору повідомлення READY\_COMMIT. Якщо він не може зафіксувати свою частину транзакції, він поміщає запис abort в файл журналу транзакцій, відправляє координатору повідомлення ABORT і відкочує свою частину транзакції (не чекаючи загального сигналу GLOBAL\_ABORT).

2. Якщо учасник відправив координатору повідомлення READY\_COMMIT, то він очікує відповіді координатора в межах встановленого тайм-ауту.

3. При отриманні GLOBAL\_ABORT учасник поміщає запис abort в файл журналу транзакцій, відкочує свою частину транзакції і відправляє координатору підтвердження відкату.

4. При отриманні GLOBAL\_COMMIT учасник поміщає запис commit в файл журналу транзакцій, фіксує свою частину транзакції і відправляє координатору підтвердження фіксації.

5. Якщо протягом встановленого тайм-ауту учасник не отримує повідомлення від координатора, він відкочує свою частину транзакції.

Якщо координатор (або учасник) не отримує протягом встановленого тайм-ауту очікуваного повідомлення, то на відповідному вузлі запускається протокол ліквідації. (Стан тайм-ауту на рис. 4.4 зазначено темним фоном.)

Протокол ліквідації для координатора:

1. Тайм-аут в стані WAITING: координатор не може зафіксувати транзакцію, бо не отримані всі підтвердження від учасників про фіксацію. Ліквідація полягає в відкаті транзакції.

2. Тайм-аут в стані DECIDED: координатор повторно розсилає відомості і прийнятому глобальному вирішенні і чекає відповідей від учасників.

1. Найпростіший протокол ліквідації для учасника полягає в блокуванні процесу доти, поки сеанс зв'язку з координатором не буде поновлено. Але з метою підвищення продуктивності (і автономності) вузлів можуть бути вжиті й інші дії: Тайм-аут в стані INITIAL: якщо не приходить команда PREPARE, то учасник не може повідомити про своє рішення координатору і не може зафіксувати транзакцію. Але він може вжити дій по відкоту транзакції. Якщо він пізніше (після закінчення тайм-ауту і відкату локальної транзакції) отримає команду PREPARE, він може проігнорувати її або відправити координатору повідомлення ABORT.

2. Тайм-аут в стані PREPARED: учасник вже сповістив координатор про прийняте ним рішення і він не може його змінити. (Якщо це рішення - відкотити локальну частину транзакції, то він виконує її не чекаючи відповіді від координатора і тому не входить в стан очікування). Але зафіксувати транзакцію він не може до отримання глобального рішення. Учасник виявляється заблокованим.

Один з можливих варіантів вирішення цієї проблеми полягає у використанні кооперативного протоколу ліквідації: учасник запрошувати інших учасників транзакції про прийняте глобальному рішенні. Найпростіший спосіб інформувати учасників про те, які ще вузли беруть участь в транзакції - передати разом з командою PREPARE список всіх учасників. При використанні цього протоколу, якщо всі учасники транзакції встановили, що сталася відмова координатора, вони можуть вибрати нового координатора і закінчити транзакцію, виконавши її відкат під управлінням нового координатора.

Один з протоколів вибору нового координатора заснований на впорядкування всіх вузлів, причому першим в цьому ланцюжку стоїть координатор. Всі вузли знають свої ідентифікатори та номери інших вузлів. Кожен вузол Si починає відправляти повідомлення узам Si + 1, Si + 2 ..., причому саме в такому порядку. Якщо вузол Sk отримує повідомлення від вузла з меншим номером, він розуміє, що не може бути новим координатором, і припиняє пересилання повідомлень. Рано чи пізно кожен з учасників дізнається, чи існує в системі вузол в меншим номером. Якщо його немає, вузол приймає рішення стати новим координатором. Якщо новообраний координатор знову потрапляє в стан тайм-ауту, протокол вибору запускається знову.

Дії, які виконуються на відмовив вузлі після його перезавантаження, називаються протоколом відновлення. Вони залежать від того, в якому стані знаходився вузол, коли стався збій, і яку роль виконував цей вузол в момент відмови: координатора або учасника.

При відмові координатора:

1. У стані INITIAL: процедура 2ФФ ще не запускалася, тому після перезавантаження слід її запустити.

2. У стані WAITING: координатор вже направив команду PREPARE, але ще не отримав усіх відповідей і не отримав жодного повідомлення ABORT. У цьому випадку він перезапускає процедуру 2ФФ.

3. У стані DECIDED: координатор вже направив учасникам глобальне рішення. Якщо після перезапуску він отримає всі підтвердження, то транзакція вважається успішно зафіксованої. В іншому випадку він повинен вдатися до протоколу ліквідації.

При відмові учасника мета протоколу відновлення - гарантувати, що після відновлення вузол виконає щодо транзакції те ж дію, яке виконали інші учасники, і зробить це незалежно від координатора, тобто по можливості без додаткових підтверджень. Розглянемо три можливих моменту виникнення відмови:

1. У стані INITIAL: учасник ще не встиг повідомить про своє рішення координатору, тому він може виконати відкат, тому координатор не міг прийняти рішення про глобальну фіксації транзакції без голосу цього учасника.

2. У стані PREPARED: учасник вже направив відомості про своє рішення координатору, тому він повинен запустити свій протокол ліквідації.

3. У стані ABORTED / COMMITED: учасник вже завершив обробку своєї частини транзакції, тому ніяких додаткових дій не потрібно.

#### Керування паралелізмом

Для керування паралелізмом в розподілених системах, так само як і в нерозподілених, використовується метод блокування. (У декількох більш сучасних системах уже використовується засіб різноманітного керування, однак у більшості систем для цього все-таки продовжує використовуватися метод блокування).

У розподіленій системі запити на перевірку, установку і зняття блокувань є повідомленнями (тут передбачається, що розглянутий об'єкт знаходиться на віддаленому вузлі), що спричиняє додаткові накладні витрати. Розглянемо, наприклад, транзакцію Т, що має потребу у відновленні об'єкта, який має репліки на n віддалених вузлах. Якщо кожен вузол керує блокуваннями для об'єктів, збережених на цьому вузлі (як це було б при дотриманні локальної автономії), то для найпростішого способу керування паралелізмом треба було б принаймні 5n повідомлень:

n запитів на блокування,

n дозволів на блокування,

n повідомлень про відновлення,

n підтверджень,

n запитів на зняття блокування.

Звичайно, можна легко удосконалити цей спосіб, використовуючи підтвердження, вкладені в блок даних зворотнього напрямку. У такий спосіб можуть комбінуватися запит на блокування і повідомлення про відновлення, а також дозвіл на блокування і підтвердження. Але навіть у такому випадку загальний час відновлення може бути на кілька порядків вище, ніж у централізованій системі.

Звичайний підхід до цієї проблеми полягає в прийнятті стратегії первинної копії, що у короткій формі була представлена вище. Для даного об'єкта R вузол, що містить первинну копію об'єкта R, буде керувати всіма операціями блокування R (пам'ятайте, що первинні копії різних об'єктів у загальному випадку можуть бути розташовані на різних вузлах). При такій стратегії множина усіх копій об'єкта з метою блокування може розглядатися як єдиний об'єкт, а загальне число повідомлень буде знижене від 5n до 2n +3 (один запит на блокування, один дозвіл на блокування, n відновлень, n підтверджень і один запит на зняття блокування). Однак зверніть увагу, що при такому рішенні знову втрачається автономність, тобто транзакція може виявитися неуспішною, якщо первинна копія недоступна (навіть при використанні транзакції тільки для читання), а локальна копія доступна. (Пам'ятайте, що для блокування первинної копії необхідні не тільки операції відновлення, а також операції вилучення. Таким чином, неприємний побічний ефект при використанні стратегії первинної копії полягає в зниженні продуктивності і доступності даних як для операцій вилучення, так і для операції відновлення.)

Іншою проблемою блокування у розподіленій системі є те, що вона може привести до глобального тупика, що охоплює два чи більше вузлів. На рисуеку представлений приклад виникнення такого тупика:

Кожна з транзакцій виконується на кожному з вузлів, при цьому її виконання відбувається в три етапи: блокування - виконання – розблокування.

Транзакція Т1 стартувала на Х, заблокувала вузол Х і виконується , але для завершення їй потрібно чекати завершення Т1 на вузлі В.

Транзакція Т1 стартувала на вузлі В, але той заблокований Т2.

Транзакція Т2 стартувала на вузлі В, заблокувала вузол У і виконується, але для завершення їй потрібно чекати завершення Т2 на вузлі Х;

Транзакція Т2 стартувала на вузлі Х, але той заблокований Т1.

Очікування завершення

виконання транзакції

Т2 на узле Х

Блокування Lx

Т1

T2

Очікування зняття блокування на вузлі

Y транкзакціею Т2

Вузол Х

Очікування зняття блокування на вузлі Х транкзакціею Т1

Вузол У

Т1

Блокування Ly

Т2

Очікування завершення

виконання транзакції Т1 на вузлі Y

Агент транзакції Т1 на вузлі Х очікує, коли закінчиться виконання транзакції Т1 на вузлі У.

Агент транзакції Т1 на вузлі В очікує, коли агент транзакції Т2 зніме блокування на вузлі У.

Агент транзакції Т2 на вузлі В очікує, коли закінчиться виконання транзакції Т2 на вузлі Х.

Агент транзакції Т2 на вузлі Х очікує, коли агент транзакції Т1 зніме блокування на вузлі Х.

**Тупикова ситуація!**

Проблема тупика такого типу полягає у тому, що жоден з вузлів не може знайти у тупик, використовуючи тільки інформацію, що зосереджена на цьому вузлі. Інакше кажучи, у локальних діаграмах чекання немає ніяких циклів, але вони з'являться при об'єднанні локальних діаграм у глобальну діаграму чекання. Звідси випливає, що виявлення глобальних тупиків зв'язано зі збільшенням накладних витрат, оскільки для цього потрібно додатково об’єднати окремі локальні діаграми.

**4.Автоматичне перетворення форматів даних**

Як тільки кілька комп'ютерів різних моделей під керуванням різних операційних систем з'єднуються в мережу, відразу виникає питання про узгодження форматів представлення даних. Дійсно, у мережі можуть бути комп'ютери, що відрізняються розрядністю (16-ти, 32-х і 64-х розрядні процесори), послідовністю байт у слові, представленням чисел із плаваючою крапкою і т.д. Задача комунікаційного сервера полягає в тому, щоб на рівні обміну даними забезпечити узгодженість форматів між віддаленим і локальним вузлами для того, щоб дані, витягнуті сервером з бази на віддаленому вузлі і передані по мережі, були правильно витлумачені прикладною програмою на локальному вузлі.

**5.Автоматична трансляція кодів**

У неоднорідному комп'ютерному середовищі при взаємодії клієнта і сервера виникає також задача трансляції кодів. Сервер може працювати з однією кодовою таблицею (наприклад, EBCDІ), клієнт - з іншою (наприклад, ASCІІ), при цьому відбувається неузгодженість трактування кодів символів. Тому, якщо на локальному вузлі використовується одна кодова таблиця, а на віддаленому - інша, то при передачі запитів по мережі і при одержанні відповідей на них необхідно забезпечити трансляцію кодів. Рішення цієї задачі також лягає на комунікаційний сервер.

Ми розглянули деталі взаємодії однієї пари "клієнт-сервер". Однак у реальному житті сервер бази даних повинен обслуговувати одночасно безліч запитів від клієнтів - отже, в один момент часу таких пар може бути кілька. І всі проблеми взаємодії, про які мова йшла вище, повинні вирішуватися комунікаційним сервером для всіх цих взаємодіючих пар. Системи з архітектурою "один-до-одного" (див. Розд. Сервер бази даних) для одночасного обслуговування сервером бази даних декількох клієнтів змушені завантажувати окремий комунікаційний сервер для кожної пари "клієнт-сервер". У результаті навантаження на операційну систему збільшується, різко зростає загальне число її процесів, що збільшує витрати обчислювальних ресурсів. Це - один з недоліків архітектури "один-до-одного". Саме тому для сучасних розподілених СКБД важливо мати багатопотоковий комунікаційний сервер, що бере на себе задачі мережної підтримки безлічі клієнтів, які одночасно звертаються до сервера. На кожному вузлі мережі він підтримує множину пар з'єднань "клієнт-сервер" і дозволяє існувати одночасно множині незалежних сеансів роботи з базами даних.

**7. Міжоперабельність (шлюзи)**

Для забезпечення незалежності від СКБД необхідно щоб СКБД на різних вузлах підтримували однаковий інтерфейс. Розглянемо приклад, на вузлі Х функціонує СКБД ІNGRESS, а на вузлі В СКБД Oracle, хоча в загальному випадку можна розглядати сполучення різних СКБД.

Припустимо, що деякий користувач U на вузлі Х бажає переглянути дані з БД на вузлі Х і з бази даних на вузлі У. За визначенням він є користувачем ІNGRESS, а тому робота з РБД для нього повинна бути схожа на роботу з БД ІNGRESS. Вказана проблема вирішується в такий спосіб: система ІNGRESS повинна мати спеціальну надбудовану програму, яка називається ***шлюзом*** і взаємодіє із системою Oracle так, щоб створювалося враження, що робота Oracle виглядає так само як і робота ІNGRESSІ.

Такий шлюз повинен виконувати наступні функції:

* реалізацію протоколу обміну інформацією між системами ІNGRESS і Oracle з врахуванням форматів даних, які використовуються у різних системах;
* обробляти довільні незаплановані твердження мови SQL для Oracle, тобто повинен містити динамічну підтримку мови SQL;
* взаємне відображення типів даних між системами ІNGRESS і Oracle (наприклад, кошмарна різноманітність способів збереження дати і часу);
* відображення діалекту мови SQL для ІNGRESS на діалект мови SQL Oracle;
* відображення інформації пересилається назад від системи Oracle до системи ІNGRESS;
* відображення каталогу Oracle у відповідну структуру користувача ІNGRESS, щоб тому була зрозуміла структура БД Oracle;
* підтримувати протоколи двофазної фіксації транзакцій для обох систем;
* підтримувати необхідне блокування даних.