УДК 004.657(045)

Анохін А.С.

ЗАСІБ КЕРУВАННЯ ДОВГОТРИВАЛИМИ ТРАНЗАКЦІЯМИ В XML-ОРІЄНТОВАНІЙ СУБД

Інститут комп'ютерних технологій
Національного авіаційного університету

Проведено огляд методів керування транзакціями і виконано проектування та розробку засобу керування довготривалими транзакціями з напівструктурованими даними в XML-орієнтованій системі управління базами даних

Вступ

Найважливішим фундаментальним поняттям в сучасних системах управління базами даних є транзакція. Транзакція – це неподільна одиниця роботи над базою даних. Виділяються чотири основні властивості транзакцій:

- атомарність
- узгодженість
- ізоляція
- довговічність.

Властивість атомарності означає, результати всіх операторів, що входять в транзакцію, відображаються в базі даних (БД), або дія цих операторів повністю відсутня.

Властивість узгодженості означає, що транзакції переводять один узгоджений стан бази даних в інший без обов'язкової підтримки узгодженості у всіх проміжних точках.

Властивість ізольованості означає, що транзакції, що виконуються, "не бачать один одного". Це означає, що, якщо навіть буде запущено безліч транзакцій, що конкурують одна з одною.

Властивість довговічності означає, що коли транзакція виконана, її оновлення зберігаються, навіть якщо в наступний момент відбудеться збій в системі [1].

Довготривалою транзакцією (ДТ) можна вважати також таку транзакцію, яка є активною довгий час, або яка лишається активною більш тривалою, ніж потрібно це для програми [2].

Постановка завдання

На сьогоднішній день методи керування транзакціями для ієрархічних СУБД можна поділити на два види. До першої групи можна віднести: ієрархічний і деревовидний протоколи синхронізації. Друга група методів грунтується на синхронізації *DOM*-операцій. Але постає питання про застосування цих методів для оперування із довготривалими транзакціями в розподільчій системі. Тому в даній праці значну увагу буде приділено вдосконаленню системи керування транзакціями із використанням *Client-Server Protocol.*

Основна частина

У природжених *XML*-СУБД всі базові механізми управління даними повинні будуватися з нуля. Це включає і механізм управління транзакціями. З одного боку, це вимагає великих зусиль, але, з іншого боку, при розробці таких механізмів можна враховувати особливості ієрархічної структури *XML*, семантику *XML*-операцій, організацію зовнішньої пам'яті і методи обробки *XML*-даних в оперативній пам'яті.

Іншим найважливішим додатком *XML*-СУБД є системи генерації статистичних звітів. Транзакції, що читають, в цьому випадку є достатньо "довгими". Їх час роботи може складати декілька годин або більш. Крім того, такі транзакції прочитують величезні масиви даних для побудови статистики.

За останні декілька років дослідниками баз даних були запропоновані різні методи синхронізації операцій над *XML*-даними. Оскільки тема ця є маловивченою, то основна увага приділена джерелам, в яких велися ці дослідження. При розгляді виділимо дві групи.

Перша група методів заснована на існуючих ієрархічних (метод гранульованих блокувань) і деревовидних протоколах синхронізації [3]. При попередньому розгляді цих методів можна подумати, що вони підходять для синхронізації операцій над *XML*-даними, оскільки відповідають ієрархічній структурі *XML*-документів. Наприклад, протокол синхронізації *XML*-транзакцій, запропонований в роботі Грабса, ґрунтується на гранульованих блокуваннях. Проте при детальному аналізу можна дійти до висновку, що дані методи не зважають повною мірою на специфіку *XML*-даних і операцій над ними, і тому їх застосування може приводити до істотного пониження паралелізму конкурентних *XML*-транзакцій.

Протокол гранульованих блокувань (ПГБ) був розроблений для синхронізації ієрархічних структур, в яких один елемент включається в інший елемент, той у свою чергу включається в третій елемент і т.д. Наприклад, транзакція T_1 може видаляти відношення R, а інша транзакція T_2 паралельно читати деякі кортежі з цього відношення. Прямим рішенням для запобігання конфлікту між T_1 і T_2 було б заблокувати всі кортежі на зміну з R транзакцією T_1. В цьому випадку при спробі читання якого-небудь кортежу транзакція T_2 намагатиметься заблокувати необхідні кортежі на читання, що приведе до конфлікту між транзакціями.

Протокол гранульованих блокувань не підходить для *XML*, оскільки в *XML* передбачається, що на всіх рівнях документа зберігаються дані, і при цьому типовою операцією є вибірка значення вузла на деякому рівні, а предки цього вузла не розглядаються. При використанні ж гранульованих захоплень завжди неявно захоплюється все піддерево, що для *XML* у багатьох випадках є надмірним.

Розглянемо ПГБ для ієрархічних структур, в яких дрібні елементи бази даних вкладені в крупніші елементи. У Деревовидному Протоколі (ДП) [4] розглядаються деревовидні структури, що утворюються за рахунок скріплення елементів між собою. Прикладом такої структури є Б-дерево. При цьому передбачається, що доступ до елементу даних A здійснюється за допомогою звернення до деякої вершини (як правило це корінь ієрархії) з подальшим переміщенням вниз по піддереву до елементу A. Цей факт дозволяє відійти від загального правила двофазного блокування. ДП формулюється наступними чотирма пунктами:

– Перший запит на блокування, що ініціюється транзакцією, може відноситися до будь-якої вершине дерева.

– Наступні запити на блокування повинні задовольнятися тільки в тому випадку, якщо транзакція володіє блокуванням вершини, батьківської по відношенню до поточної.

Операції розблокування дозволено виконувати в будь-які моменти часу.

Транзакція не має можливості повторного захоплення блокування елементу після її звільнення – навіть в тому випадку, якщо транзакція утримує блокування на батьківський елемент.

Важливою відмінною рисою ДП від ПГБ є те, що блокування деякого елементу не припускає блокування нащадків цього елементу.

ДП накладає дуже строгі обмеження на напрям переміщення по *XML*-документу: при переході від одного вузла до іншого можна переміщатися тільки зверху-вниз. В той же час в *XPath* існують осі, які дозволяють переміщатися по *XML*-документу вліво, управо, вгору і вниз від заданого вузла. Таким чином, з використанням деревовидного протоколу можна підтримувати тільки обмежену підмножину *XPath*.

Друга група методів була запропонована для синхронізації *DOM*-операцій. На сьогоднішній день існує декілька протоколів для синхронізації *DOM*-операцій: *Node*2*PL*, *NO*2*PL* і *taDOM*. Перші два протоколи було запропоновано в роботі Свена Хелмера [5], а третій протокол (і його оптимізації) в роботах Пітера Хаустіна [6 - 7]. Порівняльний аналіз всіх трьох протоколів приводиться в роботі

Пітера Хаустіна [8]. Автор цієї статті приходить до висновку, що протокол *taDOM* значно перевершує протоколи *Node*2*PL* і *NO*2*PL*, і тому далі розглянемо саме *taDOM* протокол. Але перш ніж переходити до опису цих методів, стисло опишемо *DOM*-операції.

DOM API забезпечує спосіб доступу до *XML*-документа, заснований на деревовидному представленні документа. Іншими словами, з погляду користувача *DOM API*, *XML*-документ є ациклічним направленим графом. Для роботи з ним надається набір стандартних операцій, за допомогою яких можна обходити граф вузол за вузлом уздовж ребер, додавати, видаляти або модифікувати вузли. Розрізняються три типи доступу до документа: навігаційний, доступ на зміну і через запит. Коли *DOM API* ініціалізувався для *XML*-документа, важ документ аналізується і створюється *DOM*-дерево в основній пам'яті. Цей спосіб надає швидкий доступ до вузлів.

Протокол *taDOM* В протоколі *taDOM* передбачається, що *XML*-документ представляється у вигляді Спеціального *taDOM*-Дерева.

Описаний протокол добре підходить для синхронізації паралельних *DOM*-операцій. Але в сучасних *XML*-СУБД доступ до даним, як правило, виконується за допомогою високорівневих мов запитів *XQuery*.

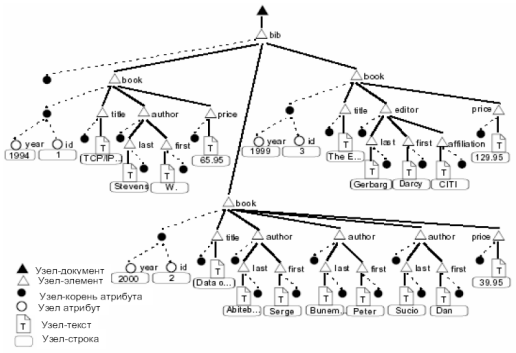

Рис. 1. *taDOM*-дерево

Крім того, описаний протокол вимагає безпосередній доступ до дерева *XML*-документа, що в деяких випадках може бути неможливим.

Застосуємо алгоритм по блокуванню транзакцій. Основна особливість цього алгоритму є специфіка умови виконання блокування транзакцій:

Для блокування ресурсів СУБД має таблицю блокувань *LOCKT* [9]. На табл. 1 зображено таблицю блокувань.

Таблиця 1. Таблиця блокувань

Ідентифікатор відношення IDENT	Режим MODE	Число транзакцій TNUMB	Список тимчасових міток транзакцій COD

У таблицю *LOCKT* заноситься інформація тільки о тих відношеннях, які використовуються у транзакціях, які виконуються у даній станції БД.

Так як при паралельному виконанні транзакцій допускається суміщення дій типу «читання», то в *LOCKT* заноситься інформація о кожній транзакції, тому створюється список транзакцій. Усі отримані вимоги на блокування поміщаються у чергу *LOCKQ*. У даному протоколі прийнята наступна стратегія блокування: у першу чергу задовольняється вимоги транзакцій з більш раднішими часовими мітками, причому якщо транзакція з самою старою міткою чекає вивільнення своїх ресурсів, ніяка інша транзакція з більш пізнішою міткою не може війти в *LOCKT*, навіть якщо її ресурси вільні.

Алгоритм блокування реалізується процесом *LOCK*:

1. Обрати з черги *LOCKQ* вимоги з самою старою міткою

2. Для неї перевірити необхідні ресурси:

Ресурси блокуються, якщо вони всі вільні, поки хоча б один з них зайнятий, транзакція не буде захоплювати інші (такий алгоритм дозволяє запізнюючій транзакції захопити свої ресурси у першу чергу).

Розблокування ресурсів по закінченню виконання транзакцій або по відкату здійснюється процедура *UNLOCK:*

3. У результаті вдоволення вимогам передати в АТС повідомлення “*LOCKED*”

4. Знищити вимогу з черги *LOCKQ*.

Роботу за цим алгоритмом можна продемонструвати на наступній схемі, зображеній на рис. 2.

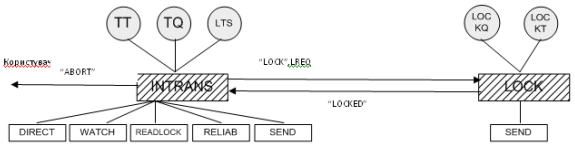

Рис. 2 Схема здійснення транзакцій

У системах для багатьох користувачів, крім того, транзакції служать для забезпечення ізольованою роботи окремих користувачів – користувачам, одночасно працюють з однією базою даних.

Розглянемо такий момент: процес буде викликано через якийсь сервіс, який пише дані у БД (в транзакції *A*) і надсилає повідомлення процесу. Процес, отримавши повідомлення, викликає інший метод цього ж сервісу, який намагається прочитати недавно записане дані з БД. І ось тут можливі два варіанти:

– Транзакція А встигла завершитися до виклику другого методу – будуть прочитані коректні дані. На рис. 3 розглянемо завершення транзакції до завершення виклику;

– Транзакція *A* не встигла завершитися до виклику другого методу – будуть прочитані некоректні дані. На рис. 4 розглянемо транзакцію, що не встигла завершитися до виклику методу.

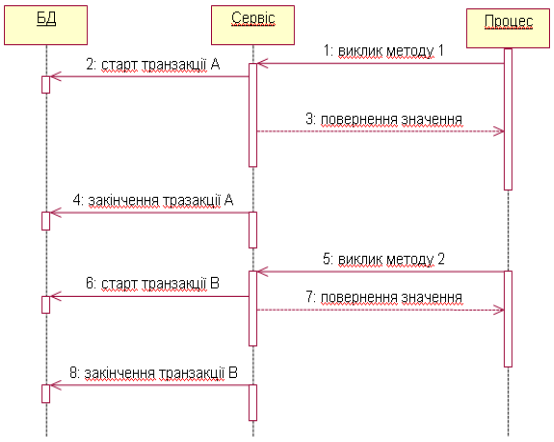

Рис. 3. Завершення транзакції до виклику другого методу

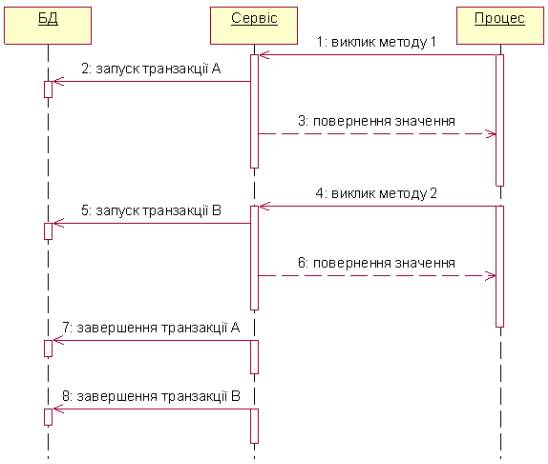

Рис. 4. Транзакція не встигла завершитися до виклику методу

У цьому й полягає основна проблема пов'язана з обробкою транзакцій, які виконують свої дії над одним і тим же елементом.

Щоб досягти результату виконання транзакцій на (рис. 5) і був розроблений засіб для керування і розподілу транзакцій в системі.

Розглянемо дві конкурентні транзакції, що виконують операції над *XML*-даними. Припустимо, що перша транзакція вибирає всі елементи *item*, у той час як друга транзакція змінює значення всіх елементів *price*. Очевидно, що транзакції не повинні конфліктувати, оскільки вони працюють з різними даними.

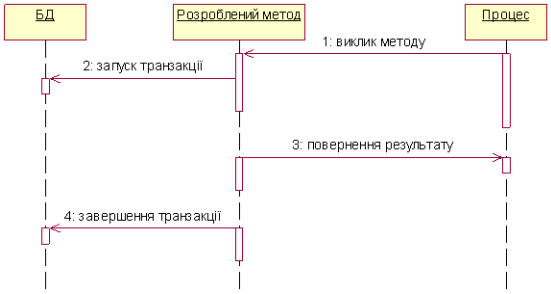

Рис. 5. Схема вдалого виконання транзакції в системі

Але планувальник транзакцій в СУБД заблокує одну з транзакцій до закінчення другої, якщо елементи *price* і *item* зберігаються в одному документі.

Необхідно відзначити, що розроблена програма працює відповідно до описаного нижче алгоритму для операції а

(op_i, t). (t позначає ідентифікатор *XML*-транзакції).

Алгоритм проведення блокування транзакцій, зміни яких призводять до зміни іншого запису.

Обчислити безліч усіх шляхів у схемі, що призводять до зміни даних, які читаються або змінюються операцією a (op_i, t_j). Назвемо цю множину *DP*.

1. Обчислити безліч вузлів n_j схеми, які відповідають кінцевим вузлам шляхів з множини *DP*. У випадку, якщо на вузол накладається предикат p_j, асоціювати його з цим вузлом. Позначимо отриману множину через $NP=\{(n_j, p_j)\}$.

2. Обчислити множину всіх вузлів *nj* схеми (та їх властивості $properties_j$), в піддереві яких можуть з'явитися фантоми. Позначимо отриману множину через $PH=\{(n_j, properties_j)\}$.

3. Якщо op_i розширює схему, то обчислити властивість $properties_i$ нового вузла.

4. Обчислити множину вузлів схеми, які зчитуються в результаті виконання операцій *locpath* в $a(op_i, t_j)$ (ця множина включає в себе вузли, прочитані *locpath* на проміжних кроках). Назвемо цю множину *RS*.

5. Для кожного $n_j \in \{RS \setminus DP\}$ встановити блокування (S, # t) і (IS, # t) на n_j та всіх предків n_j відповідно.

6. Якщо потрібного блокування встановити неможливо (воно не сумісне з вже встановленим блокуванням), то блокувати виконання операції a (op_i, t) до тих пір, поки не буде звільнене конфліктуюче блокування.

На рис. 6. зображено роботу системи з перевірки обмежень.

Для роботи з даними було використано *XML*-СУБД "*Sedna*". Це природжені *XML*-СУБД, що активно розвиваються в даний час.

Даний засіб складається з трьох суб'єктів обробки транзакцій – прикладної програми (як прикладної програми фігурує *AS*), менеджера транзакцій (*Transaction Manager – TM*) і менеджера ресурсів (*Recource Manager – RM*).

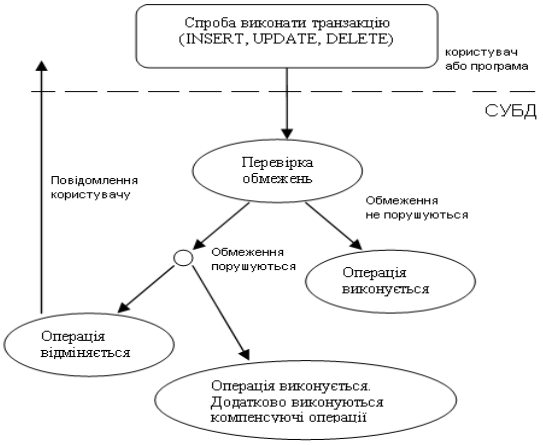

Рис. 6. Процес блокування транзакцій

На рис. 7. схематично зображемо структуру засобу керування транзакціями.

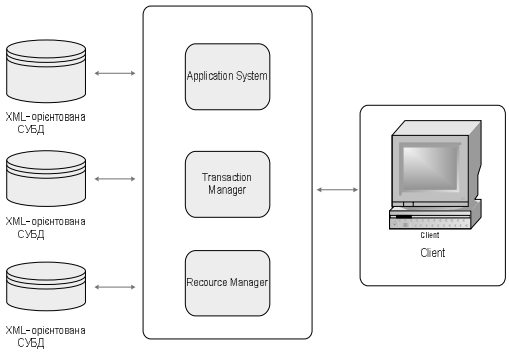

Рис. 7. Структура засобу керування транзакціями

Як видно з структури, система передбачена для роботи з декількома базами даних одночасно. Основною задачею системи є обробка запиту користувача, розбиття його на декілька під-запитів, та відправка їх відповідним СУБД.

У засобі підтримуються три види блокувань на схемі *XML*-документа: (1) структурні блокування, (2) Предикативні блокування і (3) логічні блокування.

Для кожного об'єкту транзакції сервер транзакцій автоматично створює спеціальний об'єкт, який носить назву об'єкт контексту транзакції або контекст об'єкта. Функціональність контексту забезпечується інтерфейсом *IobjectContext*.

Два методи інтерфейсу визначають спосіб виходу об'єкта з транзакції.

Метод *SetComplete* повідомляє транзакції, що він готовий до завершення своєї роботи в транзакції.

Використання методу *SetAbort* означає, що виконання коду об'єкту призвело до виникнення обставин, що перешкоджають успішному завершенню транзакції. Після використання будь-якого з цих двох методів об'єкт завершує свою участь в транзакції.

Методи *EnableCommit* і *DisableCommit* повідомляють про поточний стан об'єкту. Метод *EnableCommit* повідомляє, що об'єкт дозволяє завершити транзакцію, хоча його функціонування ще не завершено. На рис. 8 зображено засіб керування транзакцією.

Рис. 8. Засіб керування транзакцією

Виклик методу *DisableCommit* показує, що в даний момент поточний стан об'єкту не дозволяє завершити транзакцію. При спробі завершити транзакцію після виклику цього методу, транзакція буде перервана.

За допомогою перерахованих методів об'єкт контексту забезпечує інформацією про стан об'єкта транзакції.

Здатність об'єкта бути деактивованим і повторно активованим, поки клієнт зберігає посилання на нього, називається активізацією *Just-in-time*.

У процесі роботи програми часто буває необхідно використовувати один екземпляр об'єкту кілька разів через певні проміжки часу. При зверненні до об'єкта він активізується, а деякий час після припинення використання додаток утримує посилання на невикористовуваний об'єкт.

Коли створюється об'єкт, також створюється відповідний контекст об'єкту. Цей контекст об'єкта існує протягом усього часу життя відповідного об'єкта, через один або кілька циклів. Засіб використовує контекст об'єкта для збереження інформації про нього при деактівізації.

Об'єкт створюється в неактивному стані і стає активним лише після запиту клієнта.

Коли об'єкт стає неактивним, середовище знищує всі ресурси об'єкта, у тому числі, наприклад, підключення до бази даних.

Об'єкт стає неактивним при виникненні наступних подій:

– Виклик методів *SetComplete* або *SetAbort* інтерфейсу *IobjectContext*. Якщо об'єкт викликає метод *SetComplete*, коли він успішно завершив свою роботу і немає необхідності зберігати внутрішній стан об'єкта для наступного дзвінка клієнта. Якщо об'єкт викликає *SetAbort*, вказуючи на неможливість успішного завершення своєї роботи і відсутність необхідності збереження стану об'єкта. Після чого об'єкт повертається в стан, що передує цій транзакції. При нормальній реалізації об'єкта деактивізація відбувається після виклику кожного методу.

– Транзакція зберігається або переривається. Потім об'єкт також деактивізується. Серед цих об'єктів можуть продовжити своє існування тільки ті, що мають посилання на клієнтів за межами даної транзакції. Наступний виклик цих об'єктів повторно активує їх і служить причиною для виконання в такій транзакції.

Останній клієнт звільняє об'єкт. При цьому об'єкт деактивізується і контекст об'єкта теж звільняється.

В якості мови запитів використуємо мову *XQuery* [10]. *XQuery* є функціональною мовою, тому в ній вирази можуть бути вкладені одна в одну довільним чином.

У розробці використуємо підмножину мови зміни частин *XML*-документів *XUpdate*. Синтаксис цієї мови виглядає таким чином:

```
update: = 'InsertInto' '(' constr1 ',' locpath ')' |
        'InsertBefore' '(' constr2 ',' locpath ')' |
        'InsertAfter' '(' constr2 ',' locpath ')' |
        'Delete' '(' locpath ')'
            'Rename' '(' locpath, QName ')'
constr1: = 'element' '(' QName ')' content |
        'attribute' '(' QName ')' content
constr2: = 'element' '(' QName ')' content
content: = '(' PCDATA ')' | '(' ')'
```

Кожна операція приймає на вхід кілька аргументів (*constr*1, *constr*2 і *locpath*). Вирази *constr*1 і *constr*2 визначають нові вузли. Вираз *locpath*1 визначає шлях адресації вузлів у *XML*-документі, які є цільовими вузлами операції зміни. Нижче наводиться опис кожної операції.

– *InsertInto* (*constr*1, *locpath*): вставляє новий вузол (елемент або атрибут) в кожний цільовий вузол у позицію останнього дочірнього вузла.

– *InsertBefore* (*constr*2, *locpath*): вставляє новий вузол (тільки елемент) для кожного цільового вузла в позицію попереднього брата. *InsertAfter* (*constr*2, *locpath*) вставляє новий вузол (тільки елемент) для кожного цільового вузла в позицію подальшого брата.

– *Delete* (*locpath*): здійснює глибоке видалення вузлів (разом з усіма нащадками), що визначаються *locpath*.

– *Rename* (*locapth*, *QName*): присвоює нове ім'я *QName* цільовим вузлів операції.

Далі використємо позначення I_I, I_A, I_B, D і R_N для посилань на операції *InsertInto*, *InsertAfter*, *InsertBefore*, *Delete* і *Rename* відповідно. Крім того, будемо використовувати позначення *I** для посилання на довільну операцію вставки.

Для тестування створеного засобу та ініціювання транзакцій створимо клієнтську програму, що імітує процес оформлення замовлень. На головній формі програми помістимо кнопку з написом "*Connect*", три компоненти *TDCOMConnection*, пов'язані з відповідними серверами, три компоненти *TClientDataSet*, пов'язані з відповідними компонентами *TDCOMConnection*, три компоненти *TDataSource*, пов'язані з компонентами *TClientDataSet* і блокнот з двох сторінок.

Якщо не натиснути на кнопку *Connect*, дані для компонентів *TDBGrid* залишаться колишніми, у користувача є можливість спробувати повторно вибрати для замовлення вже вибрану раніше запис. У цьому випадку один з трьох серверних об'єктів буде намагатися видалити вже віддалену запис (якщо згадати текст відповідного *SQL*-запиту, вона ідентифікується значенням первинного ключа), і в цьому випадку відповідна частина транзакції не завершиться. Відбудеться відкат назад всієї розподіленої транзакції.

На рис. 9. зображено виконання транзакції.

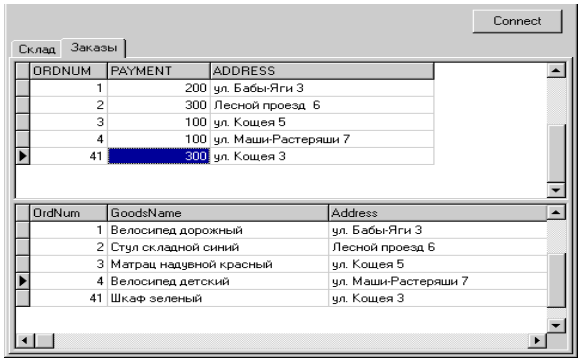

Рис. 9. Виконання транзакції

При створенні подібного роду серверних об'єктів і їхніх клієнтів повинні завжди намагатися виключити можливість помилкових дій користувача, додамо обробник подій натисканням на кнопку "Замовити":

```
procedure TForm1.Button1Click (Sender: TObject);
var n: integer; val: double; gnam, addr: widestring;
begin
try
 n: = ClientDataSet1.FieldByName ( 'GOODSNUMBER'). Value;
 val: = ClientDataSet1.FieldByName ( 'PRICE'). Value;
 gnam: = ClientDataSet1.FieldByName ( 'GOODSNAME'). Value;
 addr: = Edit1.Text;
 DcomConnection2.Connected: = true;
 DCOMConnection2.AppServer.DoTrans (n, val, addr, gnam);
 ShowMessage ( 'Замовлення прийнятий');
 Button2Click (self);
except
 ShowMessage ( 'Замовлення не прийнятий');
end;
DcomConnection2.Connected: = false; end;
```

Відзначимо також, що при натисненні на кнопку "Замовити" у разі відсутності даних у компонентах *TDBGrid* в клієнтському додатку виникне виключення, пов'язане з відсутністю потрібного поля в ком-

поненті *TClientDataSet*. Ця кнопка в такій ситуації повинна бути невибіраемой. Тому встановимо значення її властивості *Enabled* рівним *False* і перепишемо обробник події, пов'язаного з натисканням на кнопку *Connect*:

```
procedure TForm1.Button2Click (Sender: TObject);
begin
try
 DCOMCOnnection1.Connected: = true;
 DCOMCOnnection2.Connected: = true;
 DCOMCOnnection3.Connected: = true;
 CLientdataset1.data: = Dcomconnection1.Appserver.GetGoods;
 CLientdataset2.data: = Dcomconnection2.Appserver.GetPays;
 CLientdataset3.data: = Dcomconnection3.Appserver.GetDelivery;
 Button1.Enabled: = true;
except
 Button1.Enabled: = false;
 ShowMessage ( 'Один з серверних об'єктів недоступний');
end;
DCOMCOnnection1.Connected: = false;
DCOMCOnnection2.Connected: = false;
DCOMCOnnection3
```

Висновки

На сьогодні вже розроблено велика кількість методів управління транзакціями, які можна було б використовувати для управління в *XML*-СУБД. Однак можливість застосування цих методів для *XML* є відкритим дослідним питанням. Саме тому в дані праці було проведено дослідження цих методів. Для вдосконалення системи керування, що дозволяє обмінюватись даними, було розроблено засіб за рахунок використання технології *XML* у поєднанні з системою керування *XML* базами даних – "*Sedna*". Робота з СУБД ведеться через відкритий мережевий протокол *Sedna Client-Server Protocol*. Поверх цього протоколу написані драйвери (*API*) для клієнтів на *C*, *Java*, *Scheme*, *PHP*, *Python*. Дуже зручна робота з СУБД з функціонального мови *Scheme*, оскільки його основний тип – деревоподібні *S*-вирази – повністю ізоморфен (однорідний) моделі даних *XML*.

Для розробки засобу керування транзакціями використовувався мова програмування *C#* та природженою *XML* базою даних "*Sedna*".

Список літератури

1. *Кузнецов С.Д.* Основы баз данных. – 1-е изд. – М.: «Интернет-университет информационных технологий – ИНТУИТ.ру», 2005. – С. 488.

2. OASIS, Business Transaction Protocol, http://www.oasisopen/org/committees/tc_home.php?wg_abbrev=businesstransaction, 2004

3. *Gray J., Lorie R., Putzolu G.*Granularity of locks and degrees of consistency in a shared data base. Proc. VLDB Conference, 1975.

4. *Silberschatz and Z. Kedem*. Consistency in hierarchical database systems. Journal of the ACM, 27(1): 72–80, 1980.

5. *S. Helmer, C.C. Kanne, G. Moerkotte*. Lock-based protocols for cooperation on XML documents. Proc. DEXA 2003, Prague, 2003.

6. *M.P. Haustein, Theo Harder* Adjustable Transaction Isolation in XML Database Management Systems. Proc. XSym Workshop 2004, Toronto, Canada.

7. *M.P. Haustein, Theo Harder*. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. Proc. ADBIS 2003, Dresden, Germany.

8. *M. Haustein, T Harder, K. Luttenberger*. Contest of XML Lock Protocols. Proc. VLDB Conference 2006, Seoul, Korea.

9. *S. Dekeyser, J. Hidders*. Path Locks for XML Document Collaboration. Proc. WISE 2002, Singapore.

10. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 07 June 2001.http://www.w3.org/TR/2001/WD-query-datamodel-20010607.

Подано до редакції 19.04.10