**Лабораторна робота 13-14.**

**Використання транзакцій та блокувань при паралельній роботі із даними.**

**Мета роботи:** навчитись керувати транзакціями та об’єднувати у транзакції послідовності команд. Освоїти роботу із різними рівнями ізоляції транзакцій.

**Задачі:**

* Поняття транзакції та блокування.
* Режими управління транзакціями. Явний режим. Команди початку та завершення транзакції
* Обробка помилок та відкат транзакції
* Робота із вкладеними транзакціями. Фіксація та відкат на різних рівнях
* Блокування даних та рівні ізоляції транзакцій
* Поняття мертвого блокування

1. Транзакція являє собою послідовний набір команд Transact-SQL (одна або кілька команд), що утворюють логічно завершений блок, що виконується як єдине ціле. Незалежно від кількості команд в транзакції, або всі вони будуть виконані, або жодна з них не виконається. При відкаті транзакції система повертається в стан, в якому вона була до початку транзакції. Інформація про початковий стан системи зберігається в журналі транзакцій. При виконанні транзакції СКБД повинна дотримуватися правил обробки набору команд, відомих як вимоги ACID, що гарантують правильність і надійність роботи системи (Atomicity, Consistency, Isolation і Durability).

Atomicity - атомарність. Ця вимога говорить про те, що виконані в транзакції зміни будуть або виконані всі, або не будуть виконані зовсім.

Consistency - узгодженість. Всі дані після виконання транзакції повинні знаходитися в узгодженому стані, тобто всі правила і обмеження цілісності повинні бути дотримані.

Isolation - ізольованість. Зміни даних, що виконуються однією транзакцією, не повинні залежати від змін, які виконуються інший транзакцією, тобто зміни даних різними транзакціями повинні бути ізольованими.

Durability - стійкість (або довговічність). Після того як транзакція завершена, вона зберігається в системі і ніщо не може повернути систему в стан, в якому вона була до початку транзакції, тобто відбувається так звана фіксація транзакції.

Підвищення ефективності роботи невеликих транзакцій пов'язано з тим, що при виконанні транзакції сервер накладає на дані блокування, тобто забороняє іншим транзакцій звернення до даних і їх зміну до тих пір, поки не буде завершена транзакція, в контексті якої було встановлено блокування. Блокуванням (lock) називається тимчасово накладається обмеження на ви-конання деяких операцій обробки даних. Блокування можуть бути накладена як на окремий рядок таблиці, так і на всю базу даних. Управлінням блокуваннями займається менеджер блокувань (Lock Manager), контролюючий їх накладення і вирішення конфліктів. Транзакції і блокування тісно пов'язані один з одним. Транзакції накладають блокування на дані, щоб забезпечити виконання вимог ACID. Без використання блокувань кілька транзакцій могли б змінювати одні й ті ж дані.

1. SQL Server працює в перерахованих нижче режимах транзакцій: автоматичний режим - кожна окрема інструкція є транзакцією; явні транзакції - кожна транзакція явно починається з інструкції BEGIN TRANSACTION і явно закінчується інструкцією COMMIT або ROLLBACK; неявні транзакції - нова транзакція неявно починається, коли попередня транзакція завершена, але кожна транзакція явно завершується інструкцією COMMIT або ROLLBACK. Далі наведемо загальний синтаксис команд управління транзакціями в явному режимі. Почнемо із команди BEGIN TRANSACTION (рис.1)

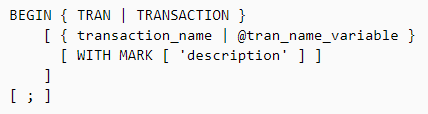


Рис.1. Синтаксис команди BEGIN TRANSACTION

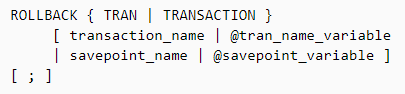
Як бачимо, при початку транзакції їх можна встановити ім’я та встановити мітку у журналі транзакцій, але цього робити не обов’язково.

Наведемо команди фіксації та відкату транзакції (рис.2-3)





Рис.2. Синтаксис команди COMMIT



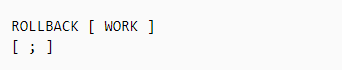


Рис.3. Синтаксис команди ROLLBACK

При виконанні фіксації та відкату транзакції також може бути вказане ім’я транзакції, що не є обов’язковим. При відкаті транзакції можливо вказання точки збереження, на яку потрібно виконати відкат транзакції. Крім того, обидві команди мають спрощену форму із необов’язковим суфіксом work.

На завершення теоретичного блоку, розглянемо синтаксис команди SAVE TRANSACTION, що дозволяє додати до тіла транзакції точку збереження (рис.4)



Рис.4. Синтаксис команди SAVE TRANSACTION

При збереженні транзакції слід вказати ім’я точки збереження для забезпечення можливості повернення транзакції на цю точку.

1. Далі перейдемо до розгляду прикладів, поступово нарощуючи їх функціональність. Для початку, реалізуємо поєднання 2-х команд в одну транзакцію за допомогою begin та commit transaction (рис.5)

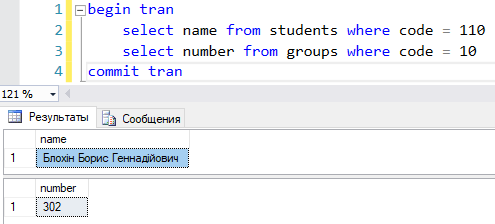


Рис.5. Поєднання декількох команд у транзакцію

Дамо нашій транзакції ім’я (рис.6)

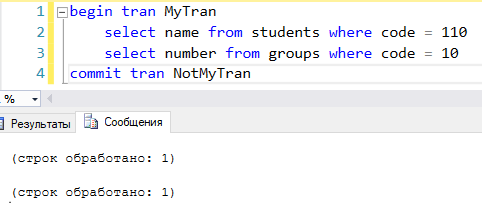


Рис.6. Початок та фіксація транзакції за її ім’ям

Як бачимо, незважаючи на те, що ми навмисно помилились в імені транзакції, помилки не виникає. Це відбувається, оскільки SQL Server фактично ігнорує ім’я в команді commit та фіксує останню відкриту транзакцію. Спробуємо зробити те ж саме із rollback (рис.7-8)

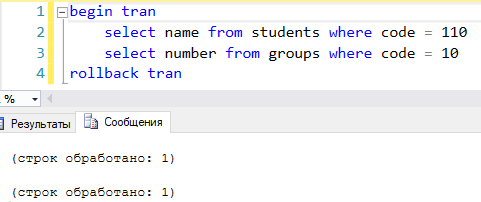


Рис.7. Використання rollback без вказання імені транзакції

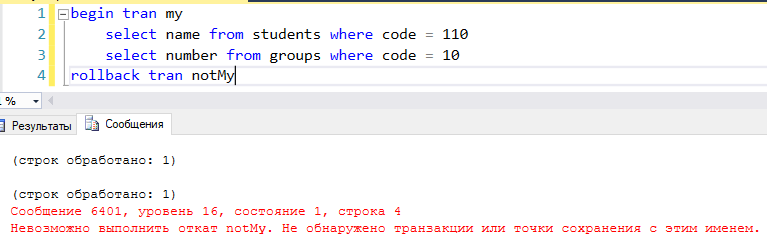


Рис.8. Використання rollback із вказанням імені транзакції

Як бачимо, команду відкату можна також застосовувати без вказання імені, але якщо ім’я вказується, воно має бути відповідним контексту.

Далі розглянемо ще один приклад фіксації та відкату транзакції, але тепер уже разом із командою зміни даних, щоб більш наочно продемонструвати різницю у поведінці сервера. Спочатку виконаємо зміну одного рядку таблиці студентів із наступним відкатом транзакції (рис.9).

У даному прикладі ми маємо 3 вибірки даних зі студентом із кодом 110. Перша вибірка до початку транзакції відображає дату народження «1998-12-30». Потім починаємо транзакцію, змінюємо дату на «1999-01-20» та повторюємо вибірку. Бачимо, що значення поля змінилося. Далі відкочуємо відкриту транзакцію, та наступною вибіркою пересвідчуємося у тому, що дані повернулися до стану, в якому були до початку транзакції, тобто дата народження знову дорівнює «1998-12-30».

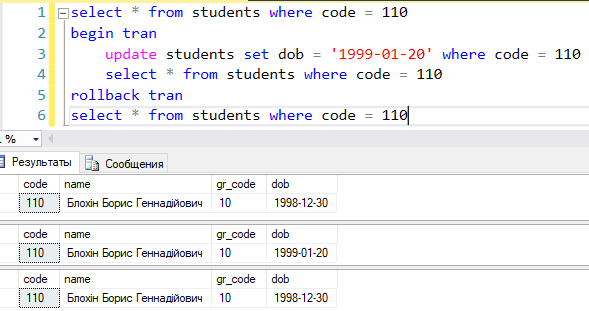


Рис.9. Зміна даних із наступним відкатом транзакції.

Виконаємо той же приклад, але тепер зафіксуємо відкриту транзакцію (рис.10)

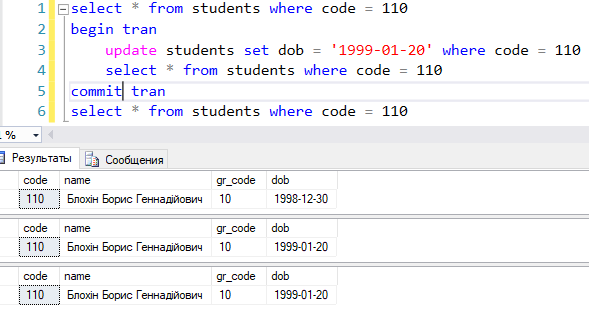


Рис.10. Зміна даних із наступною фіксацією транзакції

Перші 2 вибірки залишаються без змін, а у третій бачимо вже нове значення дати народження, оскільки транзакція була зафіксована, а отже зміни записані до БД.

1. Розглянемо ще одну можливість T-SQL – збереження транзакції і почнемо, як завжди, із наведення загального синтаксису команди SAVE TRANSACTION (рис.11)



Рис.11. Загальний синтаксис команди SAVE TRANSACTION

При виконанні команди вказуємо ім’я точки збереження, яке можемо задати явно, або із використанням змінної. Далі дана точка збереження може бути використана при відкаті транзакції, як було наведено на рис.3.

Розглянемо приклад, на якому наочно продемонструємо принцип роботи команди SAVE TRANSACTION. Для цього у межах однієї транзакції виконаємо дві команди зміни даних, розділивши їх точкою збереження, та виконаємо відкат транзакції до вказаної точки (рис.12)

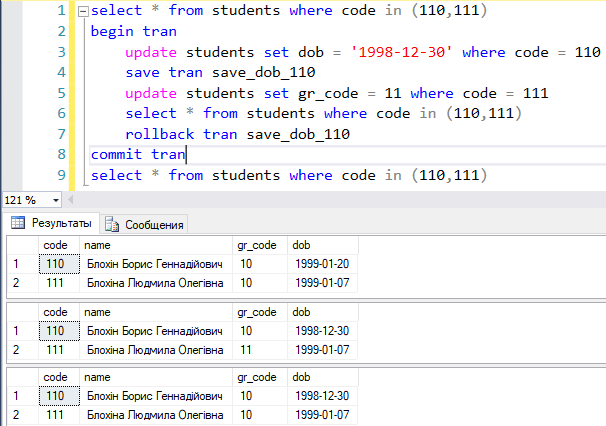


Рис.12. Використання команди SAVE TRANSACTION

Отже, як бачимо, порівнявши 1 та 2 вибірки, у межах транзакції було виконано дві зміни, після чого виконано віткат транзакції до стану, що було зафіксовано після виконання першої зміни. Відповідно після фіксації транзакції та остаточної вибірки даних видно, що перша зміна була застосована, а друга відмінена.

Розглянемо ще один приклад, що буде забезпечувати одночасне виконання 2-х команд у транзакції. Тобто, якщо хоча б одна із них поверне помилку, жодна не має бути виконана. Будемо додавати нового студента та збільшувати на 1 поле «кількість студентів» у табличці «groups» (рис.13)

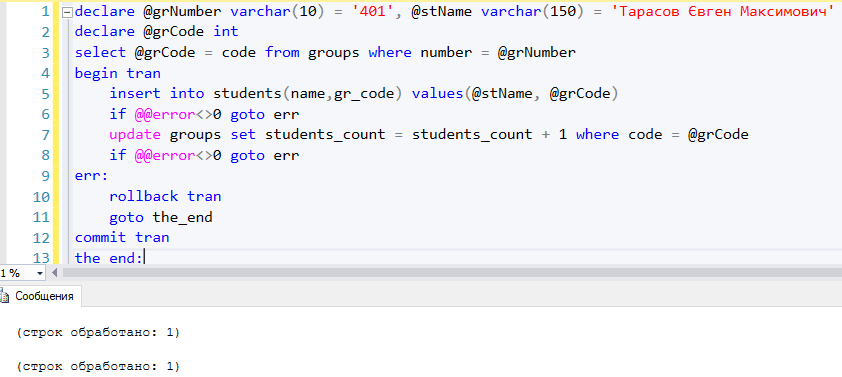


Рис.13. Обробка помилок у транзакції

Із появою у мові T-SQL конструкції try…catch реалізувати даний функціонал можна дещо простіше. Крім того, код буде мати більш зрозумілу форму (рис.14)

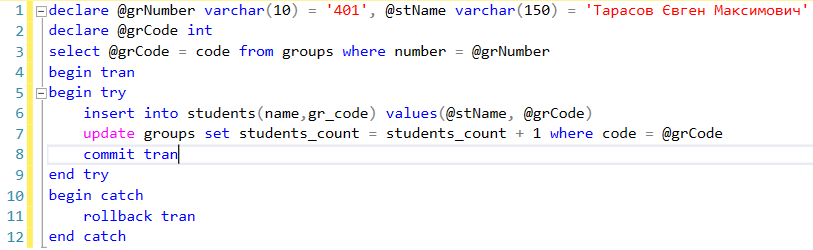


Рис.14. Обробка помилок у транзакції із використанням try…catch

1. Транзакції в T-SQL можуть бути вкладеними одна в одну. Тобто перед фіксацією або відкатом уже відкритої та активної транзакції викорується відкриття ще однієї транзакції (рис.15)

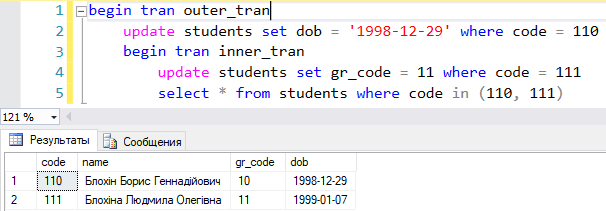


Рис.15. Приклад вкладеної транзакції

Дана ситуація найбільш розповсюджена при виконанні в тілі транзакції збереженої процедури, що також працює із транзакціями, як наприклад на рис.16-17

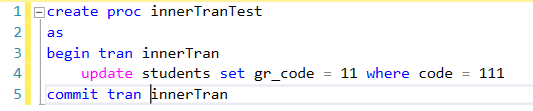


Рис.16. Збережена процедура із транзакцією.

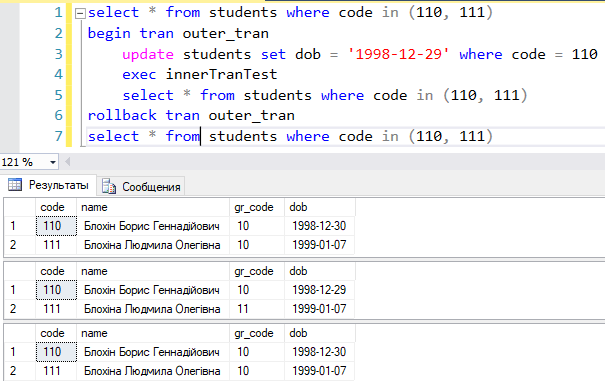
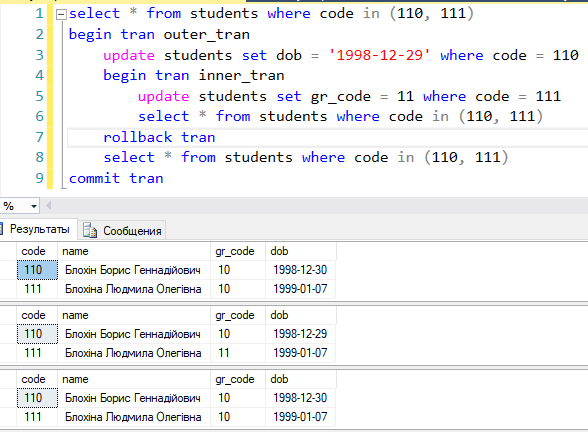


Рис.17. Вкладена транзакція із використанням збереженої процедури

На даному прикладі також показана особливість відкату верхньої транзакції при фіксації вкладеної. Як бачимо, зміни що були виконані як у зовнішній так і у внутрішній транзакції не були зафіксовані, не зважаючи на фіксацію транзакції нижнього рівня. Дана поведінка обумовлена вимогою атомарності, оскільки вкладена транзакція є частиною зовнішньої, а зовнішня є єдиним неподільним цілим із точки зору сервера, то при її відкаті будуть автоматично відкочені всі вкладені транзакції незалежно від їх попередньої фіксації.

Розглянемо зворотну ситуацію, відкотимо внутрішню транзакцію, але при цьому зафіксуємо зовнішню (рис.18)



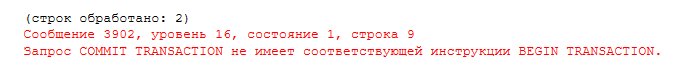


Рис.18. Фіксація зовнішньої транзакції при відкаті внутрішньої

На даному прикладі можемо звернути увагу відразу на 2 особливості. По перше, при відкаті внутрішньої транзакції відміняються не тільки зміни, виконані всередині неї, а також і зміни, що були зроблені у зовнішній транзакції. Також бачимо, що спроба фіксації зовнішньої транзакції веде до помилки. Така поведінка також обумовлена вимогою атомарності до транзакції. Якщо ми відміняємо внутрішню транзакцію, що є частиною зовнішньої, це означає, що зовнішня транзакція виконана не повністю. Виходячи із поняття атомарності такого бути не може, отже зовнішня транзакція також буде відкочена.

Насправді, команда rollback tran автоматично застосовується до транзакції найвищого рівня, тому наступна команда фіксації приводить до помилки, так як тепер немає відкритих транзакцій.

При роботі із вкладеними транзакціями також може бути використана системна змінна @@trancount, що повертає кількість відкритих транзакцій на даний момент (рис.19)

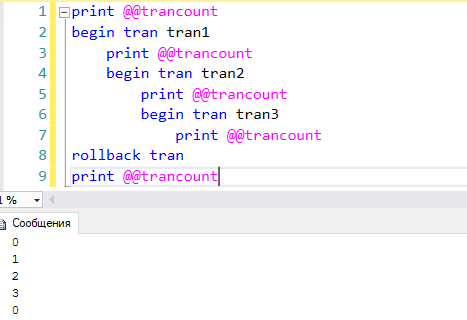


Рис.19. Системна змінна @@trancount

У прикладі рис.18 можемо додати використання @@trancount при спробі фіксації зовнішньої транзакції (рис.20). Дана перевідка запобігає виникненню помилки.

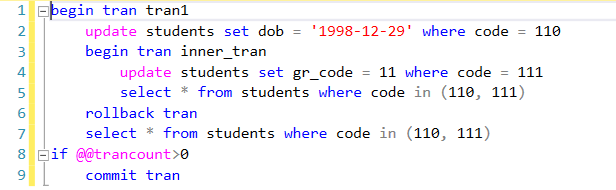


Рис.20. Перевірка наявності відкритих транзакцій перед фіксацією

1. Далі розглянемо поняття рівнів ізоляції транзакцій одна від одної та команди визначення того чи іншого рівня ізоляції. Почнемо із теоретичних відомостей.

Якщо в системі управління базами даних не реалізовані механізми блокування, то при одночасному читанні і зміні одних і тих же даних кількома користувачами можуть виникнути наступні чотири проблеми одночасного доступу:

* Проблема останньої зміни (The Lost Update Problem). Коли кілька користувачів змінюють один і той же рядок, спираючись на його початкове значення, то частину даних буде втрачено, так як кожна наступна транзакція перезапише зміни, зроблені попередньою транзакцією.
* Проблема «брудного» читання (The Uncommitted Dependency Problem). Припустимо, що користувач виконує складні операції обробки даних, що вимагають їх багаторазової зміни. Якщо під час зміни даних інший користувач буде зчитувати дані, то може виявитися, що він отримає логічно невірну інформацію. Для уникнення подібних проблем треба дочекатися закінчення зміни всіх даних, а вже потім зчитувати їх.
* Проблема не повторюваного читання (The Inconsistent Analysis Problem). Ця проблема виникає, коли транзакція зчитує одні й ті ж дані кілька разів. Під час виконання першої транзакції інша транзакція може змінити дані, так що при повторному читанні перша транзакція отримає вже інший набір даних.
* Проблема читання фантомів (The Phantom Read Problem). Ця проблема виникає, коли транзакція вибирає дані з таблиці, а інша транзакція вставляє нові рядки до завершення першої транзакції, що задовольняють умові вибірки. Якщо перша транзакція виконує складні багатокрокові зміни (або розрахунки), це може викликати невідповідність вхідних даних на різних етапах читання.

Для вирішення вищеописаних потенційних проблем за стандартом ANSI є 4 рівня ізоляції, кожен із яких вирішує всі попередні + поточний рівень проблем. Чім нижче рівень ізоляції, тим менше транзакції «заважають» одна одній, але тим більша ймовірність отримати невірні результати при виконанні тих чи інших дій. І навпаки, найвищій рівень ізоляції вирішує всі перераховані проблеми, але забороняє будь які дії із даними, з якими працює поточна транзакція.

Отже рівень ізоляції визначає ступінь незалежності транзакцій одна від одної. SQL Server підтримує чотири традиційних рівня ізоляції, заснованих на песимістичному управлінні паралелізмом (блокуванні): READ UNCOMMITTED, READ COMMITTED (використовується за умовчанням в локальних версіях SQL Server), REPEATABLE READ і SERIALIZABLE. Підтримуються також два додаткових рівня, в основі яких лежить оптимістичний підхід до паралелізму (управління версіями рядків): SNAPSHOT і READ COMMITTED SNAPSHOT (Використовується за умовчанням в SQL Database); вони є «оптимістичними» аналогами рівнів READ COMMITTED і SERIALIZABLE відповідно.

Рівень ізоляції може бути задано для всієї сесії (рис.21) або ж для одного запиту. Приклад вказання рівня ізоляції READ UNCOMMITTED для окремого запиту наведено на рис.22

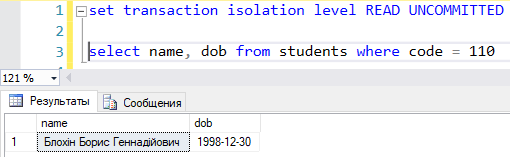


Рис.21. Задання рівня ізоляції READ UNCOMMITTED для всієї сесії

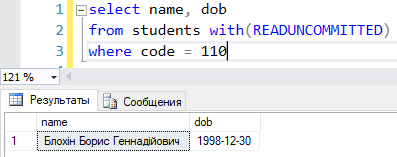


Рис.22. Задання READ UNCOMMITTED для поточного запиту

1. Далі розглянемо на прикладі роботу рівні ізоляції READ UNCOMMITTED. READ UNCOMMITTED - найнижчий рівень ізоляції, який не вимагає отримання колективного блокування для читання даних. Проілюструємо це на прикладі. У першому вікні відкриємо транзакцію виконаємо запит на зміну даних, при чому транзакцію не завершуємо (тобто на дані накладено монопольне блокування). Далі у другому та третьому вікні спробуємо отримати заблоковані дані із рівнем ізоляції READ COMMITTED (за замовченням) та READ UNCOMMITTED (брудне читання). Відкотимо транзакцію у першому вікні та повторимо дії у другому та третьому (рис.23-26)

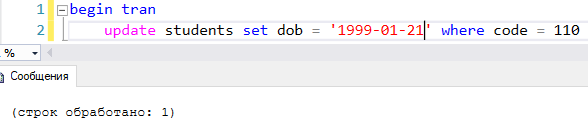


Рис.23. Початок виконання транзакції у вікні 1

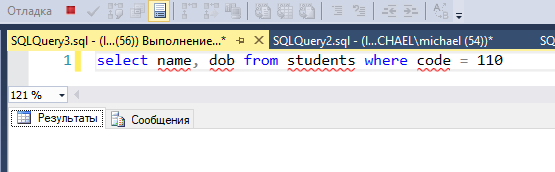


Рис.24. Спроба виконання запиту на вибірку даних у вікні 2

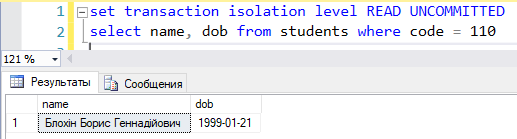


Рис.25. Виконання запиту на вибірку даних у вікні 3 із рівнем ізоляції READ UNCOMMITTED

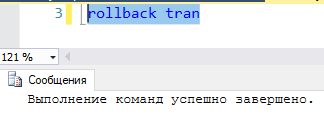


Рис.26. Відкат транзакції у вікні 1

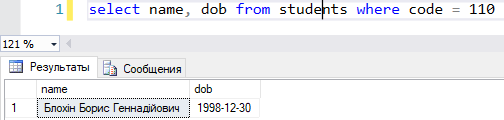


Рис.27. Завершення виконання запиту на вибірку даних у вікні 2

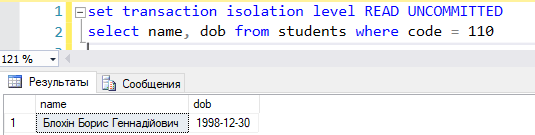


Рис.28. Повторне виконання запиту на вибірку даних у вікні 3 із рівнем ізоляції READ UNCOMMITTED

Прокоментуємо отримані результати. На рис.24 ми намагаємося вибрати студента із кодом 110, який був заблокований монопольним блокуванням транзакцією із першого вікна (рис.23). При працюємо на рівні ізоляції за замовченням (READ COMMITTED), а отже намагаємось накласти на дані колективне блокування читання даних. Це зробити неможливо, поки не зняте монопольне блокування, отже транзакція 2-го вікна очікує у черзі завершення транзакції 1-го вікна (для нас візуально запит у процесі виконання …).

Далі паралельно у 3-му вікні запускаємо такий самий запит, але із рівнем ізоляції READ UNCOMMITTED. Він не вимагає накладання жодних блокувань при читанні даних, отже вибирає потрібний нам рядок. Зверніть увагу, що значення віку студента там уже змінене, незважаючи на те, що транзакція 1-го вікна ще не зафіксована. У цьому і полягає «брудне» читання, оскільки цілком можливо, що 1-ша транзакція буде відкочена.

Зробимо це, завершимо транзакцію у 1-му вікні відкатом. Це знімає монопольне блокування, отже транзакція 2-го вікна, що очікувала у черзі, нарешті виконує запит на вибірку даних. Значення дати народження там старе, оскільки транзакцію ми відмінили. Такий самий результат дає повторне виконання запиту у вікні 3 із рівнем ізоляції READ UNCOMMITTED.

Із вищенаведеного робимо висновок, що рівень ізоляції READ UNCOMMITTED дає більш швидкий результат при читанні даних, оскільки не встановлює блокування, а отже не очікує у черзі на його встановлення. З іншого боку, дані отримані таким чином, можуть бути не зовсім достовірними, оскільки враховують непідтверджені зміни.

1. Як уже зазначалося, SQL-Server за замовченням працює на рівні ізоляції READ COMMITTED, що дозволяє отримати оптимальне співвідношення між швидкодією та достовірністю даних. Продемонструємо проблему неповторюваного читання, що може виникнути при роботі із даними. Для цього спочатку виконаємо читання даних у першій транзакції, змінимо їх у другій і повторимо читання у першій (рис. 29-31)

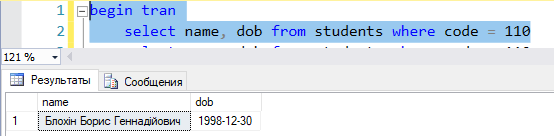


Рис.29. Перше читання даних у першій транзакції

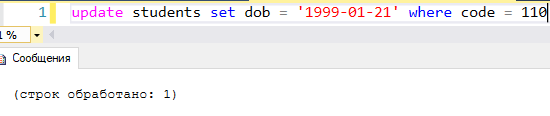


Рис.30. Зміна даних у другій транзакції

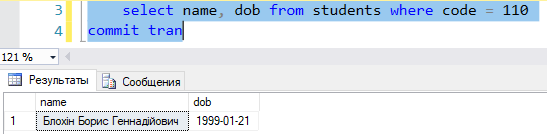


Рис.31. Повторне читання даних у першій транзакції

Як можемо бачити на рис.29 та 31 дві операції читання даних дають різні результати дати народження, незважаючи на те, що вони знаходяться у межах однієї транзакції. Дана поведінка обумовлена тип, що при використанні рівня ізоляції READ COMMITTED колективне блокування читання накладається не до завершення всієї транзакції, а лише до завершення виконання конкретного запиту.

Для вирішення даної проблеми потрібно переведення першої транзакції на наступний рівень ізоляції - REPEATABLE READ (рис.32-36)

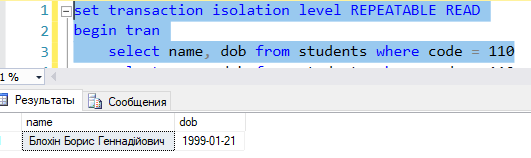


Рис.32. Початок транзакції 1 та перше читання даних

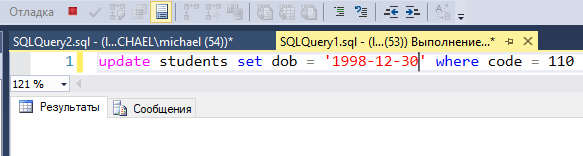


Рис.33. Транзакція 2 – спроба зміни даних

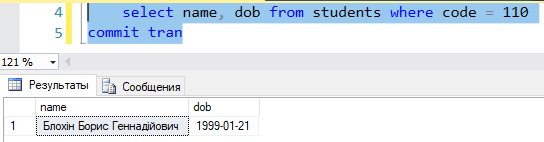


Рис.34. Повторне читання та завершення транзакції 1

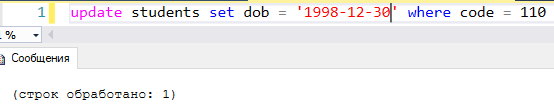


Рис.35. Завершення транзакції 2

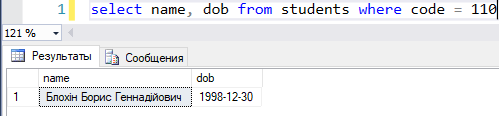


Рис.36. Вибірка даних в третьому вікні

Дамо коментарі до одержаних результатів. На рис.32 встановлюємо для транзакції 1 рівень ізоляції REPEATABLE READ та виконуємо першу операцію читання даних. В цей момент на дані накладається колективне блокування читання, яке буде знято лише по завершенню всієї транзакції 1. Далі на рис.33 транзакція 2 намагається змінити дані, для чого вона намагається накласти монопольне блокування. Це неможливо, поки є колективне блокування читання, отже транзакція 2 очікує у черзі (запит у процесі виконання …).

На рис.34 виконуємо повторне читання даних у транзакції 1 – воно дає такий самий результат, як і перше читання та завершуємо транзакцію 1. Перейшовши до вікна транзакції 2 (рис.35) бачимо, що команда зміни даних успішно виконалась, оскільки знялось колективне блокування читання транзакції 1 і транзакція 2 змогла накласти монопольне блокування, необхідне для зміни. Після цього у третьому вікні перевіримо зміну даних, виконавши вибірку, та пересвідчуємося у тому, що вона пройшла успішно (рис.36).

Отже, використання більш високого рівня ізоляції транзакції REPEATABLE READ привело до подолання проблеми неповторюваного читання, але негативно вплинуло на швидкість виконання інших транзакцій, які намагаються змінити ці дані, оскільки змушує їх стояти у черзі до завершення першої.

Інші рівні ізоляції в рамках даної роботи не розглядаються.

1. Наступне поняття, що буде розглянуто – «мертве» блокування. «Мертві», або тупикові, блокування (Deadlocks) виникають, коли дві транзакції блокують два блока даних і для завершення роботи кожної з транзакцій необхідний доступ до даних, заблокованих раніше іншою транзакцією. Для завершення кожної з транзакцій необхідно дочекатися, поки частина даних, блокована іншою транзакцією, буде розблокована. Але це неможливо, так як друга транзакція очікує розблокування ресурсів, що використовуються першою транзакцією.

Розглянемо виникнення мертвого блокування на прикладі. Для цього зробимо наступне. У першому вікні почнемо транзакцію та змінимо перший фрагмент даних. У другому також розпочнемо транзакцію, але змінимо другий фрагмент даних. Далі у першому вікні спробуємо прочитати другий фрагмент, а у другому – перший (рис.37-40)

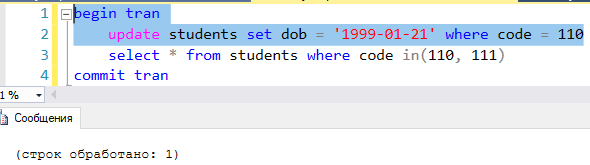


Рис.37. Початок першої транзакції та команда зміни студента із кодом 110

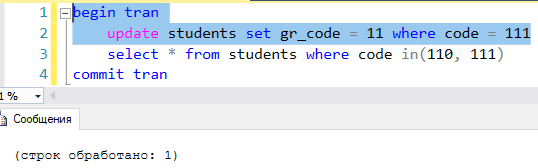


Рис.38. Початок другої транзакції та команда зміни студента із кодом 111

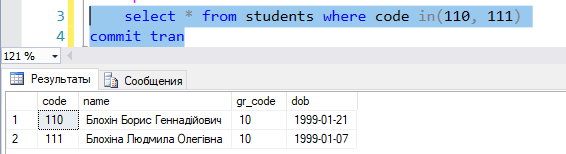


Рис.39. Вибірка студентів із кодами 110 і 111, та завершення першої транзакції

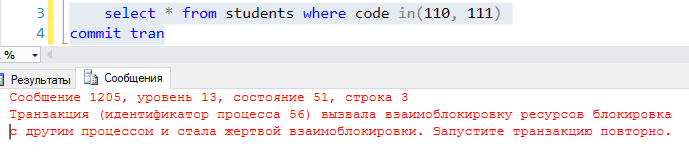


Рис.40. Вибірка студентів із кодами 110 і 111, та завершення другої транзакції

Прокоментуємо поведінку наший транзакцій. Перша транзакція починає роботу із команди зміни студента із кодом 110, тобто накладає на нього монопольне блокування до завершення транзакції. Друга транзакція робить те ж саме із студентом 111. Далі перша транзакція намагається прочитати студентів із кодами 110 та 111, але не може отримати доступ до заблокованого другою транзакцією студента (111) та стає у чергу (очікує завершення транзакції 2). Друга транзакція також намагається прочитати студентів із кодами 110 та 111, але не може отримати доступ до заблокованого першою транзакцією студента (110) та стає у чергу (очікує завершення транзакції 1).

Тобто транзакції взаємно очікують завершення одна одної. Ситуація може бути ускладнена наявністю і інших транзакцій, що також хочуть тримати доступ до студентів 110 та 111, і також будуть очікувати у черзі. За замовченням параметр lock\_timeout (максимальний час очікування у черзі) дорівнює -1, тобто необмежений, а отже транзакції будуть вічно блокувати дані рядки.

На щастя SQL Server має механізми виявлення та вирішення проблеми мертвих блокувань. При виявленні взаємно блокуючих транзакцій сервер автоматично відкочує транзакцію із меншою «вартістю», посилаючи їй повідомлення, що транзакція стала жертвою взаємного блокування (рис.40). Інша транзакція пари тепер має доступ до потрібних ресурсів та завершується успішно (рис.39).

Звернемо увагу на те, що чім вище рівень ізоляції транзакцій, тим вище ймовірність виникнення мертвих блокувань. Зменшення рівня ізоляції, навпаки, знижує ймовірність їх виникнення. Пересвідчимось у цьому, виконавши кроки із попереднього прикладу, що привели до виникнення мертвого блокування на рівні ізоляцій за замовченням (READ COMMITTED), але тепер понизимо рівень ізоляції до READ UNCOMMITTED (рис.41-44)

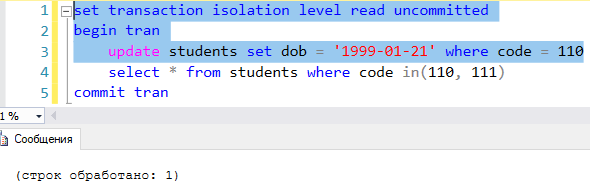


Рис.41. Початок першої транзакції із рівнем ізоляції READ UNCOMMITTED та команда зміни студента із кодом 110

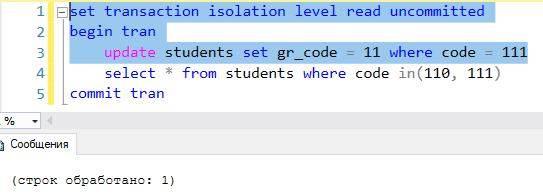


Рис.42. Початок другої транзакції із рівнем ізоляції READ UNCOMMITTED та команда зміни студента із кодом 111

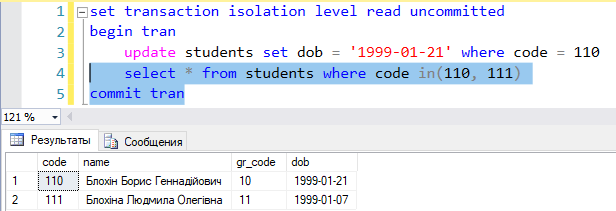


Рис.39. Вибірка студентів із кодами 110 і 111, та завершення першої транзакції із рівнем ізоляції READ UNCOMMITTED

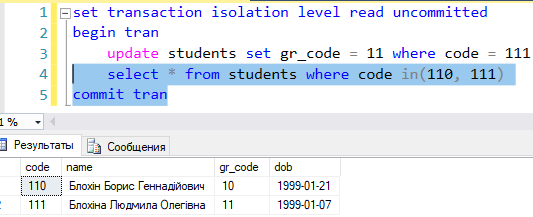


Рис.40. Вибірка студентів із кодами 110 і 111, та завершення другої транзакції із рівнем ізоляції READ UNCOMMITTED

Як бачимо, тепер мертвого блокування вдалося уникнути. Дана поведінка обумовлена відсутністю колективного блокування читання при зверненні до даних на цьому рівні ізоляції, тобто читання відбувається незважаючи на монопольне блокування зміни для цих даних.

! УВАГА. Не слід забувати також і про недоліки, пов’язані із зниженням рівня ізоляції транзакцій, оскільки у даному випадку ми читаємо непідтверджені зміни у даних.

1. Кінець роботи

**Завдання для індивідуального виконання.**

Для БД свого варіанту розробити набори команд, об’єднаних у транзакції. Приклади на базі команд зміни та вибірки даних (update та select) мають наочно демонструвати роботу команд управління транзакцією commit transaction, rollback transaction та save transaction (мінімум 3 приклади по одному на кожну команду).

На базі запитів на вибірку та зміну даних до власної БД продемонструвати використання команд управління транзакціями із використанням конструкції try…catch; показати принцип роботи із вкладеними транзакціями та особливості їх відкату та фіксації.

У власній БД змоделювати «мертве» блокування та вирішити його зміною вашого запиту, або понизивши рівень ізоляції транзакцій.

Додатково \*. Змоделювати «мертве» блокування на рівні ізоляції READ UNCOMMITTED