

Bazy Danych 2 - Lab 6

Transakcje

W bazach danych transakcja **to zbiór wykonywanych operacji, które stanowią całość**. Muszą zostać wykonane wszystkie operacje wchodzące w skład transakcji lub nie zostanie wykonana żadna z nich. Przykładem transakcji jest wykonywanie przelewu z jednego konta na drugie. Operacja przelewu musi zostać wykonana w całości, a jeżeli nie jest to możliwe, należy powrócić do stanu sprzed rozpoczęcia wykonywania operacji przelewu.

Transakcja składa się z trzech etapów:

- **rozpoczęcia**,
- **wykonania**,
- **zakończenia**.

Właściwości transakcji

Transakcja posiada następujące własności:

- **atomic** — transakcja jest niepodzielna, czyli jest wykonywana w całości lub w całości jest odwoływana.
- **consistent** — transakcja nie zmienia spójności (integralności) bazy danych, czyli wykonanie transakcji nie doprowadzi do utraty spójności danych. Jeżeli baza danych była spójna przed wykonaniem transakcji, to jest spójna również po jej zakończeniu.
- **isolated** — transakcja musi być izolowana, czyli nie może istnieć konflikt z innymi transakcjami wykonywanymi w tym samym czasie na tym samym zbiorze danych.
- **durable** — transakcja jest trwała, czyli działania wykonane w transakcji są trwałe niezależnie od tego, co się będzie działo po jej zakończeniu.

W trakcie wykonywania transakcji stosowany jest mechanizm blokowania, który gwarantuje spójność bazy danych. Oznacza to, że podczas wykonywania transakcji dane, na których jest ona wykonywana, nie ulegną zmianie lub usunięciu.

Pierwsze litery własności transakcji tworzą skrót **ACID** określający reguły, które muszą być spełnione przez serwery bazodanowe, aby transakcje mogły być wykonywane.

W serwerze SQL Server transakcja może być wykonywana na trzy sposoby:

- **Explicit — jawnie.** Rozpoczęcie transakcji jest realizowane za pomocą polecenia BEGIN TRANSACTION.
- **Autocommit — automatycznie.** Operacje wykonywane na serwerze są standardowo traktowane jako transakcje, w związku z czym nie ma potrzeby ich rozpoczynania poleceniem BEGIN TRANSACTION. Po poprawnym wykonaniu każde z poleceń jest automatycznie zatwierdzane.
- **Implicit — niejawnie.** Transakcje są wywoływanie przez programy użytkowe działające na bazie danych.

Transakcje Explicit

Transakcje Explicit są wykonywane, jeżeli zadeklarujemy chęć wykonania zapytania lub bloku zapytań w ramach transakcji za pomocą polecenia **BEGIN TRANSACTION**.

Zatwierdzenie zmian i zakończenie transakcji deklarujemy za pomocą instrukcji **COMMIT**.

Polecenie to zdejmuję blokady z tabel założone na czas trwania transakcji. Natomiast użycie instrukcji **ROLLBACK** odwołuje transakcję i odrzuca wszystkie zmiany dokonane podczas trwania transakcji. Polecenie to również zdejmuję blokady z tabel założone na czas trwania transakcji.

Przykład

Załóżmy, że od nowego roku ceny książek w księgarni internetowej wydanych po roku 2010 wzrosły o 5%, a ceny książek wydanych przed rokiem 2008 zmalały o 2%. Wykonywanie operacji zmiany cen książek zostanie zabezpieczone transakcją.

```
BEGIN TRANSACTION
UPDATE Ksiazki
SET cena = cena + cena *0.05
WHERE rok_wydania > 2010

IF @@ERROR <> 0
    BEGIN
        RAISERROR ('Błąd, Operacja zakończona niepowodzeniem !', 16, -1)
        ROLLBACK
    END

UPDATE Ksiazki
SET cena = cena - cena *0.02
WHERE rok_wydania < 2008

IF @@ERROR <> 0
    BEGIN
        RAISERROR ('Błąd, Operacja zakończona niepowodzeniem !', 16, -1)
        ROLLBACK
    END

COMMIT;
```

Pierwsza instrukcja (BEGIN TRANSACTION) uruchomiła transakcję, następna (UPDATE) przeprowadziła operację na danych. Kolejny blok to instrukcje sprawdzające, czy transakcja się powiodła. Ponieważ zmienna systemowa @@ERROR standardowo zwraca informację z numerem ostatniego błędu, może zostać użyta do sprawdzenia, czy w trakcie operacji zmiany ceny książek wystąpił błąd. Jeśli błąd nie wystąpił, zmienna ma wartość zero. Jeśli błąd wystąpił, zostanie wyświetlony komunikat (instrukcja RAISERROR) i transakcja zostanie odwołana (ROLLBACK). Jeżeli operacja przebiegła pomyślnie, przechodzimy do kolejnej operacji zmiany ceny książek.

Pojedyncza instrukcja UPDATE (dotyczy to również instrukcji DELETE i INSERT) nie musiałaby być poprzedzona uruchomieniem transakcji, ponieważ dla niej transakcja zostanie uruchomiona automatycznie. W podanym wyżej przykładzie modyfikowane dane są ze sobą wzajemnie powiązane, czyli niepowodzenie przy modyfikacji jednego zbioru danych powinno spowodować anulowanie modyfikacji drugiego zbioru danych - dlatego należy użyć transakcji.

Transakcje Autocommit

Transakcje Autocommit podobnie jak transakcje Implicit inicjuje MS SQL Server za każdym razem, gdy zostanie wydane polecenie modyfikowania danych. Ale po wykonaniu polecenia serwer automatycznie zatwierdza lub odrzuca transakcję.

Dla serwera MS SQL Server tryb automatyczny jest domyślnym trybem zarządzania transakcjami. Należy pamiętać, że jeżeli pracujemy w trybie Autocommit, każde wydane polecenie jest osobną transakcją.

Przykład

Wykorzystując funkcję systemową @@TRANCOUNT, sprawdzimy działanie trybu automatycznego transakcji. Funkcja ta zwraca liczbę otwartych w danym momencie transakcji.

```
SELECT @@TRANCOUNT;
UPDATE Ksiazki SET cena=20
WHERE rok_wydania<2008;
SELECT @@TRANCOUNT;
```

W wyniku wykonania kodu zobaczymy, że przed rozpoczęciem wykonywania instrukcji UPDATE i po jej zakończeniu nie było otwartych żadnych transakcji.

Transakcje Implicit

Transakcje Implicit inicjuje MS SQL Server, wydając niejawnie polecenie BEGIN TRANSACTIONS. Transakcje niejawne są przeprowadzane, gdy wykonywana jest jedna z następujących komend: ALTER TABLE, CREATE, DELETE, DROP, FETCH, GRANT, INSERT, OPEN, REVOKE, SELECT, TRUNCATE, UPDATE. Naszym zadaniem jest zakończenie transakcji i jej zatwierdzenie lub wycofanie. Aby serwer pracował w trybie transakcji niejawnych, należy je włączyć za pomocą polecenia:

SET IMPLICIT_TRANSACTIONS ON

Po wykonaniu tego polecenia do końca sesji będą realizowane transakcje niejawne. W celu wcześniejszego zakończenia pracy w trybie transakcji niejawnych trzeba wpisać polecenie:

SET IMPLICIT_TRANSACTIONS OFF

Ten tryb pracy może stwarzać problemy z bazą danych wtedy, gdy zapomnimy zatwierdzić lub wycofać transakcję. Pozostanie ona otwarta i będzie blokowała dostęp do danych. Jego zaletą jest możliwość wycofywania przypadkowych lub błędnych modyfikacji danych.

Przykład

Wykorzystując polecenie SET IMPLICIT_TRANSACTIONS ON, ustawimy tryb niejawnego transakcji. Następnie stosując zmienną systemową @@TRANCOUNT, sprawdzimy działanie trybu niejawnego transakcji.

```
SET IMPLICIT_TRANSACTIONS ON;
SELECT @@TRANCOUNT;
UPDATE Ksiazki SET cena=30
WHERE rok_wydania<2007;
SELECT @@TRANCOUNT;
```

Przed rozpoczęciem wykonywania instrukcji UPDATE nie było otwartych żadnych transakcji. Natomiast po jej zakończeniu pozostała rozpoczęta jedna transakcja, ponieważ nie została ona automatycznie zamknięta. Użytkownik powinien zamknąć transakcję, zatwierdzając zmiany lub je wycofując. Aby zamknąć transakcję, należy wykonać polecenie:

```
COMMIT;
SET IMPLICIT_TRANSACTIONS OFF;
```

Dopóki transakcja nie zostanie zamknięta, dostęp do tabeli Ksiazki będzie niemożliwy.

Zagnieżdżanie transakcji

W ramach już rozpoczętej transakcji można umieścić kolejną instrukcję BEGIN TRANSACTION. Wynikiem takiego działania jest zwiększenie liczby otwartych transakcji, a nie rozpoczęcie nowej transakcji.

Mechanizm zagnieżdżonych transakcji wygląda następująco: wykonanie kolejnej instrukcji BEGIN TRANSACTION zwiększa o jeden liczbę otwartych transakcji, wykonanie instrukcji COMMIT zmniejsza o jeden liczbę otwartych transakcji, natomiast wykonanie instrukcji ROLLBACK zamyka transakcję i ustawia liczbę otwartych transakcji na zero.

Przykład

```
BEGIN TRANSACTION;

SELECT @@TRANCOUNT;

UPDATE Ksiazki SET cena=10
WHERE rok_wydania<2005;

BEGIN TRANSACTION;
SELECT @@TRANCOUNT;
UPDATE Ksiazki SET cena=12
WHERE rok_wydania=2006;

BEGIN TRANSACTION;
SELECT @@TRANCOUNT;
UPDATE Ksiazki SET cena=18
WHERE rok_wydania=2007;
COMMIT;

SELECT @@TRANCOUNT;
ROLLBACK;

SELECT @@TRANCOUNT;
```

W podanym przykładzie po pierwszym odczytanie zmiennej @@TRANCOUNT otrzymamy jedną otwartą transakcję, następnie dwie, później trzy, znowu dwie i na końcu zero otwartych transakcji.

Punkty przywracania

W większości serwerów bazodanowych można wycofać nie tylko całą transakcję, ale także jej część. W tym celu należy utworzyć punkty przywracania za pomocą instrukcji **SAVE TRANSACTION**.

Przykład

```
BEGIN TRANSACTION;  
  
INSERT INTO Klient (nazwisko, imie, PESEL)  
VALUES ('Gordon', 'Michał', '79020908765');  
  
SAVE TRANSACTION P1;  
  
INSERT INTO Klient (nazwisko, imie, PESEL)  
VALUES ('Zieliński', 'Tomasz', '89110803456');  
  
ROLLBACK TRANSACTION P1;
```

W podanym przykładzie instrukcja BEGIN TRANSACTION uruchomi transakcję. Po dodaniu do tabeli Klient danych jednego klienta instrukcja SAVE TRANSACTION P1 utworzy punkt przywracania. Po dodaniu do tabeli Klient danych kolejnego klienta, transakcja zostanie odwołana, ale nie zostaną odrzucone wszystkie zmiany dokonane podczas trwania transakcji, tylko zmiany wprowadzone po zdefiniowanym punkcie przywracania. W rezultacie, mimo że transakcja została odwołana, w tabeli Klient zostaną zapisane dane klienta o nazwisku Gordon, natomiast dane klienta o nazwisku Zieliński zostaną anulowane.

Współbieżność to zdolność do jednoczesnego realizowania wielu procesów (wątków) opartych na wspólnych danych. Polega ona na przełączaniu między procesami w bardzo krótkich przedziałach czasu, co sprawia wrażenie, że procesy wykonywane są równocześnie. Ten mechanizm znajduje szerokie zastosowanie w serwerach bazodanowych, które muszą obsługiwać jednocześnie wielu użytkowników. Aby każdy z użytkowników mógł pracować tak, jakby był jedynym użytkownikiem bazy danych, konieczne jest odizolowanie operacji (transakcji) wykonywanych przez tych użytkowników.

Kontrola współbieżności

Kontrola współbieżności w serwerach bazodanowych może odbywać się na podstawie:

- **modelu optymistycznego**
- **modelu pesymistycznego**

Model optymistyczny

Model optymistyczny opiera się na założeniu, że modyfikowanie i odczytywanie tych samych danych przez różnych użytkowników jest mało prawdopodobne, chociaż nie niemożliwe. W związku z tym można przeprowadzić transakcję bez blokowania zasobów. Jedynie w sytuacji modyfikowania danych zasoby bazy są sprawdzane w celu wykrycia konfliktów.

Model pesymistyczny

Model pesymistyczny zakłada wystąpienie konfliktów, dlatego dane są blokowane za każdym razem, gdy jakaś transakcja próbuje je odczytać lub modyfikować.

Blokowanie danych

Blokowanie danych stosujemy w celu zagwarantowania integralności i spójności danych w trakcie realizowania transakcji. Dzięki temu użytkownicy nie mogą odczytywać danych, które są właśnie zmieniane przez innych użytkowników, oraz nie mogą równocześnie modyfikować tych samych danych. **Bez blokad dane znajdujące się w bazie bardzo szybko stałyby się niespójne, a definiowane na nich zapytania dawałyby błędne wyniki.**

Serwery bazodanowe automatycznie ustawiają blokady, ale na potrzeby własnej aplikacji można modyfikować ich standardowe ustawienia.

Tryby blokad

Istnieją dwa tryby blokad. Decydują one, czy możliwe jest założenie blokady na dane, które wcześniej zostały zablokowane przez inny proces.

- **Blokady współdzielone S** (ang. Shared) są zakładane domyślnie na odczytywanych obiektach tylko na czas wykonania zapytania. Jeżeli dane zostały zablokowane w trybie S, to możliwe jest założenie na nie blokady S przez inne procesy.
- **Blokady wyłączne X** (ang. eXclusive) są zakładane na modyfikowanych obiektach i domyślnie utrzymywane do zakończenia całej transakcji. Użytkownicy modyfikujący dane blokują innych użytkowników.

Zakresy blokad

Blokady mogą być zakładane na różnym poziomie szczegółowości — na poziomie wierszy, kluczy indeksów, stron, tabel, zakresów lub bazy. Dla każdej transakcji dynamicznie jest określany odpowiedni poziom, na którym należy założyć blokadę. Poziomy, na których zakładane są blokady, są ustawiane i kontrolowane przez serwer bazodanowy. To on sprawdza, czy blokady przydzielone na jednym poziomie respektują blokady ustawione na innym poziomie.

Im większe obiekty są blokowane, tym dłużej nie są one dostępne dla użytkownika (mniejsza współbieżność), ale dzięki temu zmniejsza się liczba blokad, którymi musi zarządzać serwer.

Zakleszczenia

Zakleszczenie (ang. Deadlock) powstaje, gdy jeden proces próbuje założyć blokadę powodującą konflikt z blokadą, którą próbuje założyć inny proces. W wyniku nie mogą zostać założone blokady wymagane do ukończenia rozpoczętych procesów. Serwery bazodanowe posiadają algorytmy, które automatycznie wykrywają zakleszczenia i przerywają wykonywanie jednej z transakcji.

Izolowanie transakcji

Wiemy, że transakcja musi być izolowana, czyli próba odczytania danych z tabeli, na której przeprowadzana jest transakcja przez innego użytkownika, zakończy się niepowodzeniem. Dopiero pojawienie się polecenia COMMIT powoduje właściwą modyfikację danych, zdjęcie blokad z danych i umożliwia dostęp do nich innym użytkownikom.

W zależności od istniejącego na serwerze bazodanowym poziomu izolowania transakcji może pojawić się jeden z następujących problemów:

- **Utrata aktualizacji** (ang. Lost or buried updates) — problem pojawi się, gdy dwie transakcje modyfikują te same dane. Żadna z transakcji nie wie, że druga transakcja korzysta z tych samych danych. W efekcie transakcja, która zakończy się później, zmodyfikuje dane, nisząc zmiany dokonane przez transakcję, która zakończyła się wcześniej. Domyślnie skonfigurowany serwer nie dopuści do utraty aktualizacji.
- **Brudne odczyty** (ang. Dirty reads) — problem pojawia się, gdy jedna transakcja dokonuje zmian danych, a druga w tym czasie odczytuje te dane. W efekcie transakcja odczytuje dane, które nie zostały jeszcze zatwierdzone i mogą zostać wycofane. Domyślnie skonfigurowany serwer nie dopuści do brudnych odczytów.
- **Niepowtarzalne odczyty** (ang. Non-repeatable reads) występują, gdy powtórzenie w ramach transakcji tego samego odczytu daje inny wynik. Może to być spowodowane tym, że po pierwszym odczytanie (a nie po zakończeniu transakcji) zostaną zdjęte blokady założone na odczytywane dane. Niezablokowane dane mogą zostać zmienione przez inne działania, a powtórne ich odczytanie da inny wynik. Domyślnie skonfigurowany serwer dopuszcza niepowtarzalne odczyty.
- **Odczyty widma** (ang. Phantom reads) występują, gdy pomiędzy dwoma odczytami tych samych danych w ramach jednej transakcji zmieni się liczba odczytywanych wierszy (na przykład w wyniku wykonania w międzyczasie instrukcji INSERT lub DELETE na tych danych). Domyślnie skonfigurowany serwer dopuszcza odczyty widma.

Poziomy izolowania transakcji

Blokad używamy w celu zabezpieczenia danych w trakcie współbieżnego wykonywania transakcji. Pozwala to na wykonywanie transakcji w pełnej izolacji od innych transakcji i zapewnia przez cały czas poprawność danych w bazie.

Dzięki temu możliwe jest wykonywanie kilku transakcji w tym samym czasie, tak jakby były wykonywane jedna po drugiej, czyli szeregowo.

Transakcje nie zawsze wymagają pełnej izolacji. Możemy wpływać na sposób zakładania blokad przez serwery bazodanowe, zmieniając poziom izolacji transakcji.

Poziom izolacji określa stopień, do którego dana transakcja musi być izolowana od innych transakcji. Najniższy poziom izolacji to maksymalny stopień współbieżności, ale jest on okupiony najmniejszym stopniem spójności danych. Natomiast wyższy stopień izolacji transakcji zwiększa poprawność danych, ale zmniejsza stopień współbieżności. Najwyższy poziom izolacji gwarantuje najwyższy poziom spójności danych kosztem ograniczenia do minimum współbieżności.

Serwery bazodanowe pozwalają ustawać na poziomie serwera, baz danych lub pojedynczych sesji poziom izolowania transakcji.

Standard SQL3 definiuje cztery poziomy izolacji transakcji:

- **Read Uncommitted,**
- **Read Committed,**
- **Repeatable Read,**
- **Serializable.**

Read Uncommitted

Read Uncommitted to tryb niezatwierdzonego odczytu. Jest to najniższy poziom izolacji transakcji. Odczyt danych nie powoduje założenia blokady współdzielonej. Tylko fizycznie uszkodzone dane nie będą odczytywane. Na tym poziomie pojawiają się brudne odczyty, niepowtarzalne odczyty i odczyty widma. Nie występuje jedynie problem z utratą aktualizacji. **Ten tryb może być stosowany do odczytywania danych, o których wiemy, że w czasie odczytywania nie będą modyfikowane.**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

Przykład

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
GO
SELECT miejscowosc, ulica, nr_domu
FROM Klient
WHERE nazwisko='Nowak' AND imie='Andrzej';
```

Read Committed

Read Committed to tryb odczytu zatwierdzonego. Jest to domyślny poziom izolacji stosowany przez MS SQL Server. Podczas odczytu danych zostanie założona na nie blokada współdzielona. **Założona blokada eliminuje brudne odczyty. Natomiast nadal występują niepowtarzalne odczyty oraz odczyty widma.**

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Przykład

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRANSACTION;

SELECT miejscowosc, ulica, nr_domu
FROM Klient
WHERE nazwisko='Nowak' AND imie='Andrzej';

COMMIT;
```

Repeatable Read

Repeatable Read to tryb powtarzalnego odczytu. Założona na dane blokada współdzielona jest utrzymywana aż do zakończenia całej transakcji. Dzięki temu, że inny proces nie może zmodyfikować odczytywanych danych, zostały wyeliminowane niepowtarzalne odczyty, natomiast nadal występują odczyty widma.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

Przykład

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;

SELECT nazwisko, imie
FROM Klient
WHERE miejscowosc='Gdańsk';

COMMIT;
```

Serializable

Serializable to tryb szeregowania. Transakcje odwołujące się do tych samych danych są szeregowane, czyli wykonywane jedna po drugiej. Jest to najwyższy poziom izolacji transakcji, gdzie transakcje są w pełni izolowane od siebie. Na czas trwania transakcji blokowane są całe obiekty, a nie tylko odczytywane dane, co pozwala wyeliminować odczyty widma, ale powoduje, że inni użytkownicy nie mogą modyfikować przechowywanych w obiekcie danych. Na przykład odczytując jeden wiersz tabeli, blokujemy możliwość modyfikowania danych w całej tabeli.

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Przykład

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;

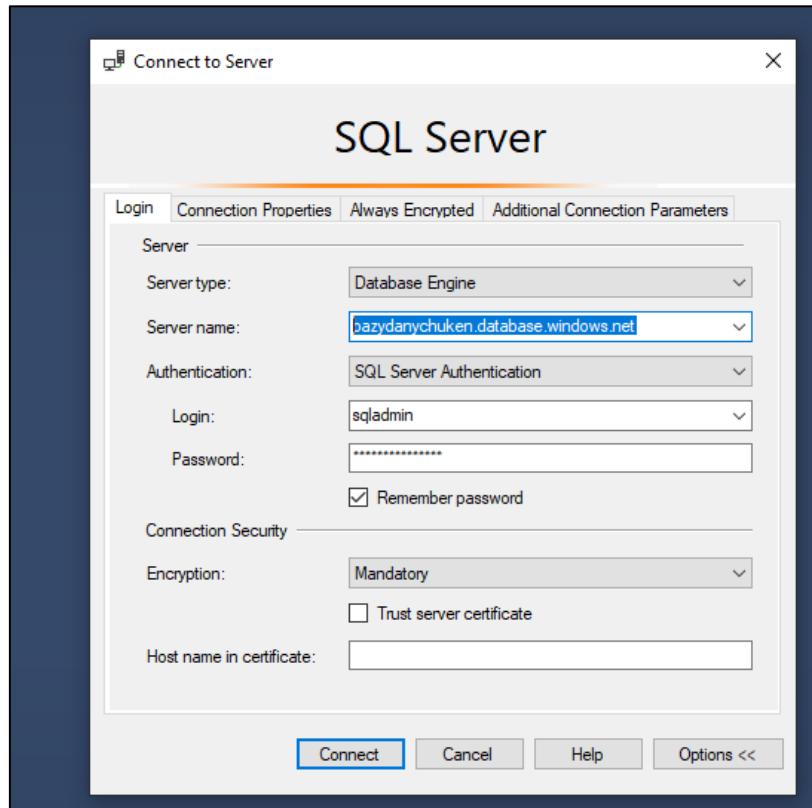
SELECT miejscowosc, ulica, nr_domu
FROM Klient
WHERE nazwisko='Adamek' AND imie='Marek';

COMMIT;
```

Materiały dodatkowe

- <https://www.geeksforgeeks.org/sql-transactions/>
- https://www.plukasiewicz.net/Artykuly/SQL_Transactions
- <https://www.cs.put.poznan.pl/mwojciechowski/slides/bd/Transakcje.pdf>
- <https://orimoladekolade.medium.com/understanding-sql-transactions-a-beginners-guide-17649bc54d50>
- <https://www.datacamp.com/tutorial/sql-transactions>
- <https://www.tutorialspoint.com/sql/sql-transactions.htm>
- <https://www.youtube.com/watch?v=Mnwow8ZoPrM>
- <https://www.youtube.com/watch?v=B0KPnFoEWIo>

Uruchom SQL Server Management Studio (<https://learn.microsoft.com/en-us/ssms/download-sql-server-management-studio-ssms>) i skonfiguruj podstawową bazę danych (https://www.youtube.com/watch?v=_12OOGKzi7I). Można wykorzystać gotową bazę danych:



Server name: bazydanychuken.database.windows.net

Login: sqladmin

Password: BazyDanych1234.!

Zadania praktyczne

Zadanie 1 – Atomowość i ROLLBACK (jedno okno)

Wybierz produkt, który na pewno istnieje:

```
SELECT ProductID, Name, ListPrice FROM SalesLT.Product WHERE ProductID = 680;
```

Pokaż atomowość:

```
BEGIN TRAN;          -- start

UPDATE SalesLT.Product

SET ListPrice = ListPrice + 100  -- +100 $

WHERE ProductID = 680;

-- podejrzyj zmianę, ale NIE zatwierdzaj

SELECT ListPrice FROM SalesLT.Product WHERE ProductID = 680;

ROLLBACK;           -- cofamy

-- sprawdź, że cena wróciła

SELECT ListPrice FROM SalesLT.Product WHERE ProductID = 680;
```

Efekt: po ROLLBACK nie pozostał żaden ślad modyfikacji.

Zadanie 2 – Dirty Read (READ UNCOMMITTED)

Session A

```
BEGIN TRAN;  
    UPDATE SalesLT.Product  
        SET ListPrice = ListPrice + 1  
    WHERE ProductID = 707; -- dowolny istniejący  
        -- nie zamykaj transakcji
```

Session B

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT ProductID, ListPrice  
FROM SalesLT.Product  
WHERE ProductID = 707; -- zobaczysz już +1, choć A nie zrobiło COMMIT
```

Session A

```
ROLLBACK;
```

4. Session B

Ponów SELECT => wraca stara wartość.

Zadanie 3 – Non-Repeatable Read (READ COMMITTED)

Session A

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN;
SELECT ListPrice FROM SalesLT.Product WHERE ProductID = 708; -- zapamiętaj
```

Session B

```
UPDATE SalesLT.Product
SET ListPrice = ListPrice + 5
WHERE ProductID = 708;
```

Session A (wciąż ta sama transakcja)

```
SELECT ListPrice FROM SalesLT.Product WHERE ProductID = 708; -- zobaczysz +5
COMMIT;
```

Eksperyment: w kroku 1 zamień na REPEATABLE READ – drugi SELECT pokaże starą cenę (anomalia zniknie).

Zadanie 4 – Savepoint i częściowy rollback

```
BEGIN TRAN;

UPDATE SalesLT.Product
SET ListPrice = ListPrice + 2
WHERE ProductID = 706;

SAVE TRAN only_first; -- punkt przywracania

UPDATE SalesLT.Product
SET ListPrice = ListPrice + 2
WHERE ProductID = 707;

-- uznajmy, że ten drugi UPDATE był błędem:
ROLLBACK TRAN only_first; -- cofnij tylko do savepointu

COMMIT; -- zatwierdź pierwszą zmianę
```

Sprawdź wartości obu produktów – 706 ma +2, 707 bez zmian.

Zadanie 5 – Powtarzalny deadlock (na samych UPDATE'ach)

Session A

```
BEGIN TRAN;  
UPDATE SalesLT.Product SET ListPrice = ListPrice + 1 WHERE ProductID = 710;
```

Session B

```
BEGIN TRAN;  
UPDATE SalesLT.Product SET ListPrice = ListPrice + 1 WHERE ProductID = 711;
```

Teraz – bez zamykania transakcji:

Session A

```
UPDATE SalesLT.Product SET ListPrice = ListPrice + 1 WHERE ProductID = 711; -- czeka
```

Session B

```
UPDATE SalesLT.Product SET ListPrice = ListPrice + 1 WHERE ProductID = 710; -- czeka
```

Po kilku sekundach SQL Server zabije jedną transakcję z błędem 1205 (deadlock victim).
ROLLBACK; w ocalającej sesji zwalnia blokady.

Zadanie 6 – Phantom Read

Session A

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN;
    SELECT COUNT(*) AS CNT
    FROM SalesLT.ProductCategory
    WHERE ParentProductCategoryID IS NULL; -- kategorie główne
-- NIE COMMITUJ
```

Session B – doda nową kategorię-fantom

```
INSERT INTO SalesLT.ProductCategory (ParentProductCategoryID, Name)
VALUES (NULL, N'NowaKategoria_ ' + CONVERT(nvarchar, SYSDATETIME()));
COMMIT;
```

Session A

```
ponów SELECT COUNT(*) ... → pojawi się +1 (phantom).
```

Session A

```
ROLLBACK;
```

Session B

```
DELETE FROM SalesLT.ProductCategory
WHERE Name LIKE N'NowaKategoria_%';
```

Bezpieczeństwo baz danych

Bezpieczeństwo baz danych obejmuje zastosowanie szerokiego wachlarza mechanizmów ochrony informacji w celu zabezpieczenia baz przed naruszeniami poufności, integralności i dostępności. W praktyce wykorzystuje się różne kategorie środków: techniczne, proceduralne/administracyjne oraz fizyczne.

Typowe zagrożenia dla systemów bazodanowych

- **Nieuprawnione lub niezamierzone działania** (lub nadużycia) wykonywane przez uprawnionych użytkowników, administratorów baz danych albo administratorów sieci/systemów, jak również przez nieuprawnionych użytkowników czy hakerów – np. nieodpowiedni dostęp do danych lub metadanych, nieautoryzowane modyfikacje programów, struktur czy konfiguracji bezpieczeństwa bazy.
- **Zainfekowanie złośliwym oprogramowaniem**, skutkujące m.in. nieuprawnionym dostępem, wyciekiem danych osobowych lub zastrzeżonych, usunięciem bądź uszkodzeniem danych albo kodu, przerwaniem działania usługi czy atakami na inne systemy.
- **Przeciążenia, ograniczenia wydajności i problemy z pojemnością**, uniemożliwiające uprawnionym użytkownikom korzystanie z bazy zgodnie z przeznaczeniem.
- **Uszkodzenia fizyczne serwerów baz danych** spowodowane pożarami lub zalaniem w serwerowni, przegrzaniem, wyładowaniami atmosferycznymi, przypadkowym rozaniem cieczy, wyładowaniami elektrostatycznymi, awariami sprzętu czy jego przestarzałością.
- **Błędy projektowe i defekty programistyczne** w samych bazach lub aplikacjach towarzyszących, powodujące luki bezpieczeństwa (np. eskalację uprawnień), utratę lub uszkodzenie danych, spadek wydajności itp.
- **Uszkodzenia lub utrata danych** na skutek wprowadzenia niepoprawnych danych lub poleceń, błędów administracyjnych, sabotażu/uszkodzeń o charakterze przestępczym.

Warstwy i rodzaje środków ochrony informacji

- Kontrola dostępu
- Audytowanie
- Uwierzytelnianie
- Szyfrowanie
- Mechanizmy integralności
- Kopie zapasowe
- Bezpieczeństwo aplikacji

Dotychczas bazy chroniono głównie zaporami i systemami wykrywania włamań w sieci. Choć te rozwiązania pozostają wartościowe, coraz ważniejsze staje się samo zabezpieczanie systemów bazodanowych oraz funkcji i danych w ich wnętrzu – zwłaszcza w czasach powszechnego dostępu do sieci, w tym Internetu. Kontrole dostępu do systemu, programów, funkcji i danych, wraz z identyfikacją użytkowników, uwierzytelnianiem i zarządzaniem uprawnieniami, są kluczowe zarówno do ograniczania, jak i rejestrowania działań uprawnionych osób. Ochrona powinna więc następować „z zewnątrz do środka” i „od środka na zewnątrz”.

Wiele organizacji opracowuje własne **minimalne standardy bezpieczeństwa** dla baz, odzwierciedlające wymagania korporacyjnych polityk bezpieczeństwa informacji, regulacji prawnych (RODO, PCI DSS itp.) oraz dobre praktyki (np. utwardzanie systemów). Projekty bezpieczeństwa określają także zarządzanie użytkownikami, analizę logów, replikację i backupy oraz biznesowe kontrole w samych programach bazy (walidację danych, ścieżki audytowe). Procedury i wytyczne uzupełniają te rozwiązania o ręczne kontrole.

Uprawnienia

- **Uprawnienia systemowe** – pozwalają lokalnemu użytkownikowi wykonywać działania administracyjne.
- **Uprawnienia do obiektów** – umożliwiają operacje na konkretnych obiektach bazy, przyznane przez innego użytkownika (np. USAGE, SELECT, INSERT, UPDATE, REFERENCES).

Zasada najmniejszych przywilejów

Bazy podlegające kontrolom wewnętrznym (np. raportowanie finansowe) wymagają **rozdziału obowiązków**: kod powinien być przeglądany przez osobę niezależną, a programista nie może samodzielnie wdrażać zmian na produkcji. Uprawnienia produkcyjne należy ograniczyć do absolutnego minimum; ewentualne podwyższone uprawnienia przyznawać tymczasowo (np. EXECUTE AS, sudo). DBA – będąc strażnikiem danych – musi przestrzegać przepisów i reguł.

Ocena podatności i zgodność

Jedną z technik oceny bezpieczeństwa jest **test podatności/penetracyjny** bazy: próby obejścia zabezpieczeń, przejęcia systemu itp. Administratorzy używają skanerów do wykrywania błędnych konfiguracji i znanych luk, a wyniki służą do „utwardzania” bazy. W systemach krytycznych prowadzi się **ciągłe monitorowanie zgodności**, obejmujące zarządzanie łatkami i przegląd uprawnień. Program zgodności to ciągła ocena ryzyka; skany podatności są wstępem do niego.

Abstrakcja

Mechanizmy uwierzytelniania i autoryzacji na poziomie aplikacji (np. single sign-on) mogą ukryć szczegóły bazy – użytkownik loguje się raz, a system przekazuje poświadczenia dalej.

Monitorowanie aktywności bazy (DAM)

Dodatkową warstwę zapewnia **monitorowanie w czasie rzeczywistym**: analiza ruchu SQL w sieci lub agent na serwerze. Agenty rejestrują działania także administratorów, a informacje trafiają do bezpiecznego systemu, którego DBA nie może zmodyfikować. Analiza wykrywa znane ataki lub anomalie; niektóre systemy potrafią rozłączać sesje i quarantannować użytkowników. Ważna jest tu **separacja obowiązków** (SOD): DBA nie może wyłączyć DAM.

Audit natywny

Większość baz posiada własne ślady audytowe. Z uwagi na wydajność i ryzyko manipulacji eksportuje się je do systemu bezpieczeństwa, do którego DBA nie ma dostępu. Lepszą pewność (forensic) dają jednak narzędzia sieciowe lub moduły jądra.

Procesy i procedury

Program bezpieczeństwa obejmuje regularny przegląd uprawnień, dwuetapowe uwierzytelnianie użytkowników, bezpieczne przechowywanie haseł procesów automatycznych. **Plan odzwierczania awaryjnego** (DR) – np. replikacja do innego regionu – zapewnia dostępność w razie incydentu [5]. Po zdarzeniu stosuje się **informatykę śledczą** w celu ustalenia zakresu naruszenia i wdrożenia poprawek systemowych.

Materiały dodatkowe

- https://dbc.wroc.pl/Content/3017/PDF/Chalon_ochrona.pdf
- <https://documents.uow.edu.au/~jrg/115/lectures/19dbsecurity/19dbsecurity.pdf>
- <https://cwkx.github.io/data/teaching/cybersecurity/lecture5.pdf>
- <https://balaza.gitlab.io/bps5r/images/CH05-CompSec2e.pdf>
- <https://www.tutorialspoint.com/database-security>
- <https://www.ibm.com/think/topics/database-security>
- <https://www.ibm.com/think/topics/database-security>
- <https://learn.microsoft.com/en-us/azure/azure-sql/database/secure-database-tutorial?view=azuresql>
- <https://www.cisecurity.org/cis-benchmarks>

Zadania Praktyczne

Zadanie 1 - Zasada najmniejszych przywilejów

Cel: utworzyć użytkownika, który może tylko czytać listę produktów.

Utwórz login i użytkownika bazy i nadaj uprawnienia

```
USE adventureWorksSample;

CREATE LOGIN DemoReader WITH PASSWORD = 'Str0ng!Passw0rd';
CREATE USER DemoReader FOR LOGIN DemoReader;

GRANT SELECT ON OBJECT::SalesLT.Product TO DemoReader; -- kluczowe: OBJECT::
```

Próba zapisu – zaloguj się jako *DemoReader* i uruchom:

```
EXECUTE AS USER = 'DemoReader';
SELECT TOP (3) ProductID, Name, ListPrice FROM SalesLT.Product; -- działa
UPDATE SalesLT.Product SET ListPrice = ListPrice + 1 WHERE ProductID = 680; -- powinien zwrócić błąd 229 (brak uprawnień)
REVERT;
```

Zadanie 2 - Audyt natywny w SQL Server / Azure SQL

Cel: rejestrować wszystkie próby modyfikacji cen produktów.

```
-- 1. Utwórz specyfikację audytu (file log w katalogu domyślnym)
```

```
CREATE SERVER AUDIT Audit_PriceChange
```

```
    TO FILE (FILEPATH = 'C:\SqlAudit\', MAXSIZE = 10 MB);
```

```
ALTER SERVER AUDIT Audit_PriceChange WITH (STATE = ON);
```

```
-- 2. Specyfikacja bazy
```

```
CREATE DATABASE AUDIT SPECIFICATION Audit_ProductPrice
```

```
FOR SERVER AUDIT Audit_PriceChange
```

```
    ADD (UPDATE ON SalesLT.Product BY PUBLIC);
```

```
ALTER DATABASE AUDIT SPECIFICATION Audit_ProductPrice ENABLE;
```

Zmień cenę w innym oknie → sprawdź plik audytu w SSMS: Management → Audit Logs.

Zadanie 3 Prosty test SQL Injection + poprawka

Cel: zobaczyć lukę i naprawić ją parametryzacją.

Wrażliwa procedura

```
CREATE OR ALTER PROC dbo.FindCustomer @Cust nvarchar(4000)
AS
DECLARE @sql nvarchar(max) =
N'SELECT CustomerID, CompanyName FROM SalesLT.Customer WHERE
CompanyName LIKE ''' + @Cust + N'%'''';
EXEC(@sql);
GO
```

Atak

```
EXEC dbo.FindCustomer N'%"; DROP TABLE SalesLT.Product; --';
```

(Nie wykona się, bo brak uprawnień – ale znakomicie widać, jak łatwo byłoby uszkodzić bazę).

Poprawiona wersja

```
CREATE OR ALTER PROC dbo.FindCustomer @Cust nvarchar(4000)
AS
SELECT CustomerID, CompanyName
FROM SalesLT.Customer
WHERE CompanyName LIKE @Cust + N'%';
```

Zadanie 4 Szyfrowanie połączenia (TLS)

Cel: sprawdzić, czy sesja korzysta z TLS.

```
SELECT encrypt_option, client_net_address, local_net_address  
FROM sys.dm_exec_connections  
WHERE session_id = @@SPID;
```

Jeśli encrypt_option = TRUE, ruch jest szyfrowany; jeśli FALSE – włącz TLS w łączaniu (checkbox “Encrypt connection” lub parametr Encrypt=yes).

Zadanie 5 Backup i weryfikacja odtwarzania (zadanie do zrobienia lokalnie niestety na Azure się to nie uda 😞)

Cel: zrobić kopię, skasować wiersz, przywrócić kopię do nowej bazy i sprawdzić, że dane wróciły.

-- 1. Kopia

```
BACKUP DATABASE AdventureWorksLT TO DISK = 'C:\Backups\AWLT.bak' WITH INIT;
```

-- 2. Sabotaż testowy

```
DELETE FROM SalesLT.Product WHERE ProductID = 680;
```

-- 3. Przywrócenie na bok

```
RESTORE DATABASE AWLT_Restore FROM DISK = 'C:\Backups\AWLT.bak'  
WITH MOVE 'AdventureWorksLT_Data' TO 'C:\Backups\AWLT_Data.mdf',  
MOVE 'AdventureWorksLT_Log' TO 'C:\Backups\AWLT_Log.ldf',  
REPLACE;
```

-- 4. Sprawdź w AWLT_Restore, że ProductID 680 istnieje.

Zadanie 6 Monitorowanie aktywności (wykonać na bazie master)

Cel: wykryć nieudane logowania.

```
SELECT *  
FROM sys.event_log  
WHERE event_type = 'connection_failed'
```