PHP i Oracle. Tworzenie aplikacji webowych: od przetwarzania danych po Ajaksa

Autor: Yuli Vasiliev
Tłumaczenie: Robert Górczyński, Artur Przybyła
ISBN: 978-83-246-1974-0
Tytuł oryginału: PHP Oracle Web Development: Data processing, Security, Caching, XML, Web Services, and Ajax
Format: 170x230, stron: 392

Poznaj niezwykłe możliwości duetu Oracle–PHP i twórz niezawodne aplikacje!

- Jak połączyć PHP i Oracle w celu uzyskania optymalnej wydajności i niezawodności?
- Jak wykorzystywać funkcje XML w PHP i Oracle?
- Jak poprawić wydajność dzięki zastosowaniu buforowania?

Baza Danych Oracle nie ma sobie równych pod względem wydajności, niezawodności oraz skalowalności. Natomiast skryptowy język PHP dzięki niezwykłej prostocie stosowania stanowi jedno z najpopularniejszych narzędzi budowania aplikacji sieciowych – nawet dla niezbyt doświadczonych programistów. Budowanie i wdrażanie aplikacji PHP opartych na Oracle pozwala więc na optymalne połączenie potężnych możliwości i solidności z łatwością użycia i krótkim czasem programowania.

Książka „PHP i Oracle. Tworzenie aplikacji webowych: od przetwarzania danych po Ajaksa" zawiera zilustrowany praktycznymi przykładami opis technologii oraz wszystkich narzędzi potrzebnych, aby optymalnie wykorzystać możliwości duetu Oracle–PHP. Dzięki temu podręcznikowi poznasz nowe funkcje PHP i bazy danych Oracle; dowiesz się także, na czym polega programowanie procedur składowanych i obsługa transakcji. Nauczysz się tworzyć niezawodne aplikacje i zapewniać im wyższą wydajność dzięki mechanizmom buforowania, a także używać technologii Ajax z technologiami Oracle Database i funkcjami PHP w celu usprawnienia reakcji aplikacji na działania użytkownika.

- Połączenie PHP i Oracle
- Przetwarzanie danych
- Tworzenie i wywoływanie wyzwalaczy
- Używanie podprogramów składowanych
- Podejście zorientowane obiektowo
- Obsługa wyjątków
- Bezpieczeństwo
- Buforowanie
- Aplikacje oparte na XML
- Usługi sieciowe
- Aplikacje oparte na Ajaksie

Połącz wydajność, skalowalność i niezawodność z łatwością użycia i krótkim czasem programowania!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

4

ROZDZIAŁ

Transakcje

Aby uzyskać pewność, że używane dane zawsze będą prawidłowe, należy stosować transakcje. W skrócie: dostarczają one mechanizm pozwalający na bezpieczne modyfikowanie danych przechowywanych w bazie danych poprzez przeniesienie bazy danych z jednego spójnego stanu do kolejnego.

Klasycznym przykładem wykorzystania transakcji jest operacja bankowa, taka jak przelew środków pieniężnych z jednego konta bankowego na inne. Załóżmy, że zachodzi potrzeba przelania środków pieniężnych z konta oszczędnościowego na konto bieżące. W celu wykonania tej operacji trzeba będzie przeprowadzić przynajmniej dwa kroki: zmniejszyć wartości środków na koncie oszczędnościowym i zwiększyć wartość środków na koncie bieżącym. Oczywiste jest, że w tego rodzaju sytuacji konieczne będzie potraktowanie obu operacji jako pojedynczej, aby zachować saldo między kontami. Dlatego też żadna z wymienionych operacji nie może zostać przeprowadzona oddzielnie — muszą być zakończone obie lub żadna z nich — programista musi zagwarantować, że albo obie operacje zakończą się powodzeniem, albo żadna z nich nie będzie przeprowadzona. W takiej sytuacji doskonałym rozwiązaniem jest zastosowanie transakcji.

W rozdziale zostały omówione różne mechanizmy, które mogą być użyte do przeprowadzania transakcji za pomocą technologii PHP i Oracle. Zaczniemy od ogólnego omówienia transakcji, ponieważ te informacje są bardzo ważne w celu dokładnego zrozumienia sposobu działania transakcji. Następnie zostały przedstawione szczegółowy dotyczące stosowania na różne sposoby transakcji w aplikacjach PHP/Oracle.

Ogólny opis transakcji

Przed rozpoczęciem budowania własnych aplikacji PHP/Oracle wykorzystujących transakcje należy zapoznać się z podstawowymi informacjami, które dotyczą transakcji, i przekonać się, jak mogą być przeprowadzane z poziomu PHP i Oracle. W podrozdziale zaprezentowano ogólny opis transakcji oraz poruszono następujące zagadnienia:

- Czym są transakcje i kiedy programista może chcieć je stosować?
- Jak przeprowadzać transakcje za pomocą PHP i Oracle?
- Jak zorganizować aplikację PHP/Oracle, aby efektywnie kontrolować transakcje?

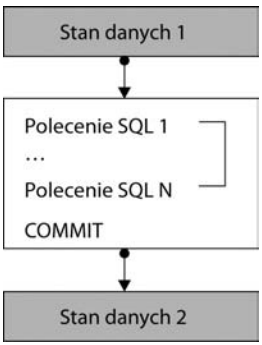
Ponieważ wymienione powyżej zagadnienia najlepiej można zrozumieć za pomocą przykładów, w podrozdziale znajdzie się kilka prostych przykładów pokazujących możliwy sposób wykorzystania transakcji w aplikacjach PHP/Oracle.

Czym jest transakcja?

Ogólnie rzecz biorąc, transakcja jest czynnością lub serią czynności, które przenoszą system z jednego stanu spójności do kolejnego. Z punktu widzenia programisty budującego aplikacje oparte na bazie danych transakcja może być uznawana za niewidzialny zestaw operacji, które przenoszą bazę danych z jednego stanu spójności do kolejnego.

> Transakcja jest jednostką logiczną pracy, zawierającą jedno lub większą liczbę poleceń SQL, które mogą być w całości albo zatwierdzone, albo wycofane.

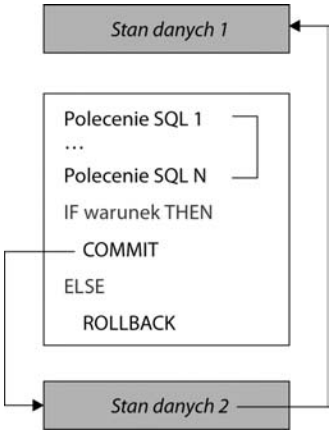
Oznacza to, że wszystkie polecenia SQL zawarte w transakcji muszą być z powodzeniem zakończone, aby cała transakcja mogła zostać zatwierdzona, dzięki czemu zmiany przeprowadzone przez wszystkie operacje DML zostają trwale przeprowadzone. Graficznie zostało to pokazane na rysunku 4.1.



Rysunek 4.1. Graficzne przedstawienie sposobu działania transakcji

Jak można zobaczyć na powyższym rysunku, polecenia SQL składające się na transakcję przenoszą dane, na których operują z jednego stanu spójności do kolejnego. Transakcja musi zostać zatwierdzona, aby wprowadzone przez nią zmiany zostały zastosowane w bazie danych i tym samym przeniosły dane do kolejnego stanu spójności. W przeciwnym razie wszystkie polecenia SQL wykonane przez transakcję zostaną wycofane, a dane pozostaną w stanie, w którym znajdowały się w chwili rozpoczęcia transakcji.

Jeżeli w trakcie wykonywania transakcji wystąpi błąd serwera, na przykład awaria sprzętu komputerowego, efekty transakcji zostaną automatycznie wycofane. Jednak w pewnych sytuacjach programista może chcieć ręcznie wycofać ukończoną (ale jeszcze nie zatwierdzoną) transakcję, w zależności od ustalonego warunku. Taka sytuacja została pokazana graficznie na rysunku 4.2.



Rysunek 4.2. Graficzne przedstawienie warunkowego zatwierdzenia transakcji

Jak widać na powyższym rysunku, po zakończeniu wykonywania wszystkich poleceń transakcji programista ma możliwość albo jej zatwierdzenia, albo wycofania.

Czym są reguły ACID?

ACID to akronim oznaczający *Atomicity* (niepodzielność), *Consistency* (spójność), *Isolation* (izolacja) oraz *Durability* (trwałość). Każdy system zarządzania bazą danych obsługujący transakcje musi spełniać powyższe cechy charakterystyczne. Ich podsumowanie znalazło się w pierwszej tabeli na następnej stronie.

Baza danych Oracle obsługuje wszystkie właściwości ACID wymienione w powyższej tabeli. Dlatego podczas budowania aplikacji transakcyjnych na Oracle nie trzeba projektować własnych schematów gwarantujących spójność i integralność danych. Zamiast tego zawsze lepszym rozwiązaniem będzie wykorzystanie transakcji Oracle i zrzucenie obsługi tego rodzaju problemów na bazę danych.

Właściwość	Opis
Niepodzielność	Transakcja stanowi niepodzielną jednostkę pracy. Oznacza to, że albo wszystkie operacje w ramach transakcji zostaną wykonane, albo nie będzie wykonana żadna z nich.
Spójność	Transakcja przenosi bazę danych z jednego stanu spójności do kolejnego. Oznacza to, że podczas przeprowadzania transakcji nie mogą być złamane żadne czynniki wpływające na spójność bazy danych. Jeżeli transakcja złamie jakiekolwiek reguły spójności, zostanie wycofana.
Izolacja	Do chwili zatwierdzenia transakcji zmiany wprowadzane przez operacje składające się na transakcję nie powinny być widoczne dla innych, równocześnie przeprowadzanych transakcji.
Trwałość	Gdy transakcja zostanie zatwierdzona, wszystkie modyfikacje wprowadzone przez transakcję staną się trwałe i widoczne dla innych transakcji. Trwałość gwarantuje, że jeśli zatwierdzenie transakcji zakończy się powodzeniem, w przypadku awarii systemu nie będą one wycofane.

W jaki sposób transakcje działają w Oracle?

W podrozdziale zostanie pokrótce przedstawiony ogólny opis działania transakcji w Oracle. Szczegółowe informacje dotyczące sposobu działania transakcji w bazie danych Oracle można znaleźć w dokumentacji Oracle: w rozdziale „Transaction Management" w podręczniku użytkownika *Oracle Database Concepts*.

W Oracle transakcja nie rozpoczyna się jawnie, lecz niejawnie podczas uruchamiania pierwszego wykonywalnego polecenia SQL. Jednak istnieje kilka sytuacji powodujących zakończenie transakcji. Przedstawiono w poniższej tabeli:

Sytuacja	Opis
Wydanie polecenia `COMMIT`	Po wydaniu polecenia `COMMIT` następuje zakończenie wykonywania bieżącej transakcji. Polecenie powoduje, że zmiany wprowadzone przez polecenia SQL transakcji stają się trwałe.
Wydanie polecenia `ROLLBACK`	Po wydaniu polecenia `ROLLBACK` następuje zakończenie wykonywania bieżącej transakcji. Polecenie to powoduje, że zmiany wprowadzone przez polecenia SQL transakcji zostają wycofane.
Wydanie polecenia `DDL`	Jeżeli zostanie wydane polecenie `DDL`, Oracle w pierwszej kolejności zatwierdzi bieżącą transakcję, a następnie wykona i zatwierdzi polecenie `DDL` w nowej transakcji, która składa się z pojedynczego polecenia.
Zamknięcie połączenia	Kiedy połączenie zostanie zamknięte, Oracle automatycznie zatwierdzi bieżącą transakcję w tym połączeniu.
Nieprawidłowe przerwanie wykonywania programu	Jeżeli wykonywanie programu zostanie nieprawidłowo przerwane, Oracle automatycznie wycofa bieżącą transakcję.

Jak można się przekonać na podstawie powyższej tabeli, transakcja zawsze będzie albo zatwierdzona, albo wycofana, niezależnie od tego, czy zostanie wyraźnie zatwierdzona, czy wycofana.

Warto jednak zwrócić uwagę, że zawsze dobrą praktyką jest wyraźne zatwierdzanie lub wycofywanie transakcji zamiast polegania na zachowaniu domyślnym Oracle. W rzeczywistości zachowanie domyślne aplikacji transakcyjnej może być różne w zależności od narzędzia stosowanego przez aplikację do nawiązywania połączenia z bazą danych Oracle.

Na przykład jeżeli skrypt PHP współpracuje z bazą danych Oracle za pomocą rozszerzenia OCI8, nie można liczyć na to, że aktywna transakcja w połączeniu zostanie automatycznie zatwierdzona po zamknięciu tego połączenia. W takim przypadku po zamknięciu połączenia lub zakończeniu wykonywania skryptu aktywna transakcja zostanie wycofana.

Natomiast jeżeli rozłączenie z bazą danych nastąpi po wydaniu polecenia `DISCONNECT` z poziomu narzędzia SQL*Plus lub nastąpi połączenie z bazą jako inny użytkownik używający polecenia `CONNECT` bądź sesja SQL*Plus zostanie zamknięta w wyniku wydania polecenia `EXIT`, aktywna transakcja w połączeniu będzie zatwierdzona.

> Aby uniknąć nieoczekiwanego zachowania w aplikacji, zawsze dobrze jest wyraźnie zatwierdzać bądź wycofywać transakcje, zamiast polegać na zachowaniu domyślnym aplikacji transakcyjnej.

Używanie transakcji w aplikacjach PHP/Oracle

Jak wspomniano w poprzednim podrozdziale, w Oracle można wyraźnie albo zatwierdzić, albo wycofać transakcję, używając polecenia odpowiednio `COMMIT` lub `ROLLBACK`. W celu wykonania powyższych poleceń z poziomu kodu PHP nie trzeba używać funkcji `oci_parse()` i `oci_execute()`, jak podczas wykonywania innych poleceń SQL, takich jak `SELECT` lub `INSERT`. Zamiast tego używa się funkcji rozszerzenia OCI8 o nazwach `oci_commit()` i `oci_rollback()`.

Przedstawiony poniżej skrypt PHP demonstruje sposób wyraźnego zatwierdzenia lub wycofania transakcji z poziomu kodu PHP podczas używania operacji DML. Zadaniem skryptu jest próba uaktualnienia rekordów tabeli `employees`, reprezentujących pracowników, których identyfikator stanowiska wynosi `ST_MAN` (menedżer magazynu). Uaktualnienie polega na zwiększeniu wysokości pensji o 10%. Jeżeli uaktualnienie jednego lub większej liczby rekordów zakończy się niepowodzeniem, cała transakcja zostanie wycofana, a uaktualnionym polom pensji będą przywrócone ich wartości początkowe. Cały proces podsumowują poniższe kroki:

- Krok 1.: wykonanie zapytania względem tabeli `employees` w celu uzyskania liczby rekordów przedstawiających pracowników na wskazanym stanowisku (menedżerów magazynu).
- Krok 2.: rozpoczęcie transakcji i wykonanie operacji `UPDATE` względem tabeli `employees` w celu przeprowadzenia próby zwiększenia pensji menedżerów magazynu o 10%.

- Krok 3.: wycofanie transakcji, jeśli liczba rekordów zmodyfikowanych przez operację UPDATE będzie mniejsza niż liczba rekordów przedstawiających menedżerów. W przeciwnym razie transakcja zostanie zatwierdzona.

Teraz warto zapoznać się z kodem źródłowym skryptu, aby przekonać się, jak powyższe kroki mogą być zaimplementowane w PHP za pomocą funkcji rozszerzenia OCI8.

```
<?php
   // Plik: trans.php.
   if(!$dbConn = oci_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   $query = "SELECT count(*) num_rows FROM employees
      WHERE job_id='ST_MAN'";
   $stmt = oci_parse($dbConn,$query);
   if (!oci_execute($stmt)) {
      $err = oci_error($stmt);
      trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
         . $err['message'], E_USER_ERROR);
   };
   oci_fetch($stmt);
   $numrows = oci_result($stmt, 'NUM_ROWS');
   oci_free_statement($stmt);
   $query = "UPDATE employees e
      SET salary = salary*1.1
      WHERE e.job_id='ST_MAN' AND salary*1.1
      BETWEEN (SELECT min_salary FROM jobs j WHERE j.job_id=e.job_id)
      AND (SELECT max_salary FROM jobs j WHERE j.job_id=e.job_id)";
   $stmt = oci_parse($dbConn,$query);
   if (!oci_execute($stmt, OCI_DEFAULT)) {
      $err = oci_error($stmt);
      trigger_error('Operacja uaktualnienia zakończyła się niepowodzeniem: '
         . $err['message'], E_USER_ERROR);
   }
   $updrows = oci_num_rows($stmt);
   print "Próba uaktualnienia ".$numrows." rekordów.<br />";
   print "Udało się uaktualnić ".$updrows." rekordów.<br />";
   if ($updrows<$numrows) {
      if (!oci_rollback($dbConn)) {
         $err = oci_error($dbConn);
         trigger_error('Nie udało się wycofać transakcji: '
            .$err['message'], E_USER_ERROR);
      }
      print "Transakcja została wycofana.";
   } else {
      if (!oci_commit($dbConn)) {
         $err = oci_error($dbConn);
```

```
            trigger_error('Nie udało się zatwierdzić transakcji: '
               .$err['message'], E_USER_ERROR);
        }
        print "Transakcja została zatwierdzona.";
    }
?>
```

W powyższym skrypcie zdefiniowano zapytanie zwracające liczbę rekordów, które przedstawiają pracowników zatrudnionych jako menedżerowie magazynu. W zapytaniu użyto funkcji `count()` w celu obliczenia liczby rekordów spełniających kryteria zdefiniowane w klauzuli `WHERE` zapytania. W omawianym przypadku wartością zwrotną funkcji `count(*)` jest liczba rekordów reprezentujących pracowników, dla których wartość pola `job_id` wynosi `ST_MAN`.

W skrypcie liczba rekordów przedstawiających menedżerów magazynu jest pobierana z bufora wynikowego za pomocą połączenia funkcji `oci_fetch()` i `oci_result()`. Nie trzeba w tym przypadku stosować pętli, ponieważ zapytanie zwraca tylko jeden rekord zawierający pojedyncze pole o nazwie `num_rows`.

Następnie wykonywane jest zapytanie uaktualniające kolumnę `salary` tabeli `employees`. Uaktualnienie polega na zwiększeniu pensji menedżerów magazynu o 10%. Zapytanie uaktualnia wysokość pensji tylko wtedy, gdy nowa wysokość pensji będzie mieściła się między minimalną i maksymalną wysokością pensji ustalonymi dla menedżerów magazynu i zdefiniowanymi w tabeli `jobs`.

W omawianym przykładzie polecenie `UPDATE` jest wykonywane w trybie wykonywania `OCI_`
`↪DEFAULT`. W ten sposób następuje rozpoczęcie transakcji, która pozwala programiście w dalszej części skryptu na wyraźne zatwierdzenie lub wycofanie zmian wprowadzonych przez polecenie `UPDATE`. Interesującą kwestią, na którą warto zwrócić uwagę, jest fakt, że domyślny tryb wykonywania to `OCI_COMMIT_ON_SUCCESS`, w którym polecenie jest zatwierdzane automatycznie, jeśli jego wykonywanie zakończy się powodzeniem.

> Według dokumentacji Oracle aplikacja zawsze powinna wyraźnie zatwierdzać bądź wycofywać transakcję przed zakończeniem działania programu. Jednak podczas używania rozszerzenia PHP OCI8 nie trzeba tego robić, gdy polecenia SQL są wykonywane w trybie `OCI_COMMIT_ON_SUCCESS`. W wymienionym trybie polecenie SQL jest zatwierdzane automatycznie, jeśli jego wykonanie zakończy się powodzeniem (czyli w sytuacji podobnej do wyraźnego zatwierdzenia natychmiast po wykonaniu polecenia). Jeżeli błąd serwera uniemożliwi zakończenie powodzeniem wykonywania polecenia SQL, Oracle automatycznie wycofa wszystkie zmiany wprowadzone przez nie.

W omawianym skrypcie funkcja `oci_num_rows()` jest wywoływana w celu uzyskania liczby rekordów zmodyfikowanych przez operację `UPDATE`. Po poznaniu liczby rekordów przedstawiających menedżerów magazynu oraz liczby faktycznie uaktualnionych można porównać te wartości i sprawdzić, czy są identyczne.

W powyższym skrypcie wycofanie transakcji następuje, gdy liczba uaktualnionych rekordów jest mniejsza niż całkowita liczba rekordów przedstawiających menedżerów magazynu. Ma to sens, ponieważ niedopuszczalna jest sytuacja, w której uaktualnione zostaną rekordy tylko niektórych menedżerów magazynu.

Możliwość wycofania zmian w przypadku wystąpienia takiej sytuacji ma znaczenie krytyczne — pozwala wówczas na użycie innego skryptu, który będzie mógł prawidłowo uaktualnić wszystkie rekordy przedstawiające menedżerów magazynu. Na przykład w rzeczywistej sytuacji prawdopodobnie pożądanym rozwiązaniem będzie zwiększenie pensji menedżera magazynu do maksymalnej dopuszczalnej wysokości, jeśli jej podwyżka o 10% przekroczy to dozwolone maksimum.

Jeżeli operacja `UPDATE` zmodyfikuje wszystkie rekordy przedstawiające menedżerów magazynu, transakcję można zatwierdzić za pomocą funkcji `oci_commit()`, dzięki czemu wprowadzone zmiany będą trwałe.

Innym elementem, na który warto zwrócić uwagę w omawianym skrypcie, jest użyty mechanizm obsługi błędów. Jeżeli w trakcie wykonywania funkcji `oci_rollback()` lub `oci_commit()` wystąpi błąd, identyfikator połączenia będzie przekazany funkcji `oci_error()`, która z kolei zwróci komunikat błędu opisujący powstały błąd.

Strukturyzacja aplikacji PHP/Oracle w celu nadzorowania transakcji

Jak Czytelnik może przypomnieć sobie z lektury rozdziału 3., ogólnie rzecz biorąc, dobrym pomysłem jest umieszczenie wewnątrz bazy danych kluczowej logiki biznesowej aplikacji PHP/Oracle. Padło tam też również stwierdzenie, że w prostych przypadkach nie trzeba nawet tworzyć kodu PL/SQL w celu przeniesienia do bazy danych procesu przetwarzania danych. Zamiast tego można zaprojektować skomplikowane zapytania SQL, które po wydaniu będą nakazywały serwerowi bazy danych przeprowadzenie wszystkich wymaganych operacji przetwarzania danych.

Wracając do przykładu omówionego w poprzednim podrozdziale: istnieje możliwość modyfikacji użytego w nim polecenia `UPDATE` w taki sposób, aby rekordy przedstawiające menedżerów magazynu były uaktualniane tylko wtedy, gdy nowa wysokość pensji każdego menedżera nadal będzie znajdowała się między wartością minimalną i maksymalną dla tego stanowiska. Wymienione wartości są zdefiniowane w tabeli `jobs`. Taka modyfikacja wyeliminuje potrzebę wykonywania oddzielnego zapytania, które zwraca liczbę rekordów spełniających powyższy warunek. W ten sposób nastąpi zmniejszenie ilości kodu koniecznego do napisania w celu implementacji żądanego zachowania funkcji.

W skrócie: nowa wersja polecenia `UPDATE` łączy w ramach pojedynczego polecenia wszystkie trzy kroki wymienione na początku poprzedniego podrozdziału. Nie trzeba nawet wyraźnie

zatwierdzać bądź wycofywać operacji UPDATE. Zamiast tego polecenie UPDATE można wydać w trybie OCI_COMMIT_ON_SUCCESS, który gwarantuje, że operacja zostanie automatycznie zatwierdzona, jeśli jej wykonanie zakończy się powodzeniem, a w przeciwnym razie — wycofana.

Przedstawiony poniżej skrypt prezentuje w działaniu nową wersję polecenia UPDATE:

```
<?php
   // Plik: transQuery.php.
   if(!$dbConn = oci_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   $jobno = 'ST_MAN';
   $query = "
      UPDATE (SELECT salary, job_id FROM employees WHERE
         (SELECT count(*) FROM employees WHERE job_id=:jobid AND
         salary*1.1 BETWEEN (SELECT min_salary FROM jobs WHERE
            job_id=:jobid) AND
            (SELECT max_salary FROM jobs WHERE
               job_id=:jobid)) IN
         (SELECT count(*) FROM employees WHERE job_id=:jobid)
      ) emp
      SET emp.salary = salary*1.1
      WHERE emp.job_id=:jobid";
   $stmt = oci_parse($dbConn,$query);
   oci_bind_by_name($stmt, ':jobid', $jobno);
   if (!oci_execute($stmt, OCI_COMMIT_ON_SUCCESS)) {
      $err = oci_error($stmt);
      trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
         . $err['message'], E_USER_ERROR);
   };
   $updrows = oci_num_rows($stmt);
   if ($updrows>0) {
      print "Transakcja została zatwierdzona.";
   } else {
      print "Transakcja została wycofana.";
   }
?>
```

W powyższym fragmencie kodu zdefiniowano polecenie UPDATE, które spowoduje uaktualnienie wszystkich rekordów reprezentujących menedżerów magazynu poprzez zwiększenie wysokości ich pensji o 10%, pod warunkiem że żadna z nowych wartości pensji nie przekroczy maksymalnej pensji menedżera magazynu zdefiniowanej w tabeli jobs. Jeżeli chociaż jedna pensja przekroczy wartość maksymalną, polecenie UPDATE nie uaktualni żadnych rekordów.

Aby uzyskać taki sposób działania, zamiast podawania tabeli employees w klauzuli polecenia UPDATE użyte zostało polecenie SELECT, którego wartością zwrotną są albo wszystkie rekordy

tabeli employees, albo żaden z nich. Zależy to od tego, czy wszystkie rekordy spełniające warunek zdefiniowany w klauzuli WHERE polecenia UPDATE (w omawianym przypadku będą to wszystkie rekordy reprezentujące menedżerów magazynu) mogą być uaktualnione w taki sposób, aby każda nowa wysokość pensji nie przekraczała wartości maksymalnej pensji dla tego stanowiska.

> To polecenie SELECT jest uznawane za widok wewnętrzny. W przeciwieństwie do zwykłych widoków omówionych w podrozdziale „Wykorzystanie zalet widoków", znajdującym się w rozdziale 3., widoki wewnętrzne nie są obiektami schematu bazy danych, ale podzapytaniami, które mogą być stosowane jedynie w ramach zawierających je poleceń za pomocą aliasów.

W omawianym przykładzie zastosowanie w poleceniu UPDATE wewnętrznego widoku emp powoduje wyeliminowanie potrzeby wykonywania oddzielnego zapytania zwracającego liczbę rekordów reprezentujących menedżerów magazynu, a następnie sprawdzającego, czy otrzymana liczba jest równa liczbie rekordów faktycznie zmodyfikowanych przez polecenie UPDATE. Teraz skrypt wykonuje tylko pojedyncze zapytanie SQL, dzięki czemu wyraźnie skraca czas wykonywania skryptu.

> Omówiony skrypt jest dobrym przykładem pokazującym korzyści, jakie można odnieść po przeniesieniu kluczowej logiki biznesowej aplikacji PHP/Oracle z PHP do bazy danych Oracle. Tu zamiast używania dwóch oddzielnych poleceń i analizowania ich wyników w PHP zastosowane zostało tylko jedno polecenie SQL, które powoduje, że przetwarzanie danych zachodzi całkowicie w serwerze bazy danych.

Warto także zwrócić uwagę na użyty w skrypcie sposób wiązania zmiennych. Zmienna PHP jobno jest dowiązywana do znacznika jobid użytego w poleceniu UPDATE. Interesujący jest fakt, że znacznik zmiennej wiązanej jobid pojawia się w poleceniu częściej niż tylko jednokrotnie.

Inaczej niż w poprzednim przykładzie, w którym polecenie UPDATE było wykonywane w trybie OCI_DEFAULT, wyraźnie rozpoczynającym transakcję, nowa wersja skryptu wykonuje polecenie w trybie OCI_COMMIT_ON_SUCCESS. Operacja UPDATE jest więc automatycznie zatwierdzana, jeśli jej wykonanie zakończy się powodzeniem.

> Jak wcześniej wspomniano, OCI_COMMIT_ON_SUCCESS jest trybem domyślnym wykonywania polecenia. Oznacza to, że nie trzeba wyraźnie go określać podczas wywoływania funkcji oci_execute(). W omawianym przykładzie został wyraźnie umieszczony w kodzie źródłowym, aby zwrócić na to uwagę.

W poprzednim przykładzie nadal używano funkcji oci_num_rows() w celu pobrania liczby rekordów zmodyfikowanych przez polecenie UPDATE. Jednak tym razem nie trzeba porównywać tej liczby z całkowitą liczbą rekordów reprezentujących menedżerów magazynu, jak to miało miejsce w przypadku poprzedniego skryptu. Wszystko, co trzeba zrobić, to prostu sprawdzić, czy liczba rekordów zmodyfikowanych przez polecenie UPDATE jest większa niż 0.

Jeżeli liczba uaktualnionych rekordów jest większa niż 0, oznacza to, że operacja UPDATE zmodyfikowała wszystkie rekordy reprezentujące menedżerów magazynu i została z powodzeniem zatwierdzona. W takim przypadku należy wyświetlić użytkownikowi komunikat informujący o zatwierdzeniu transakcji.

Jeżeli liczba uaktualnionych rekordów wynosi 0, oznacza to, że operacja UPDATE nie zmodyfikowała żadnych rekordów. W takim przypadku należy wyświetlić użytkownikowi komunikat informujący o wycofaniu transakcji. Jednak w rzeczywistości transakcja jest zatwierdzona, ale żaden rekord nie został zmodyfikowany przez operację UPDATE.

Tworzenie kodu transakcyjnego

Dotychczas przedstawiono kilka prostych przykładów pokazujących podstawy działania transakcji Oracle z PHP. W podrozdziale zostaną zaprezentowane bardziej skomplikowane przykłady stosowania transakcji w aplikacjach PHP/Oracle.

Nadzorowanie transakcji z poziomu PHP

Jak Czytelnik dowiedział się z przykładów omówionych we wcześniejszej części rozdziału, funkcja oci_execute() pozwala na wykonywanie polecenia SQL w dwóch trybach — OCI_COMMIT_
↪ON_SUCCESS oraz OCI_DEFAULT.

Podczas gdy stosowanie trybu OCI_COMMIT_ON_SUCCESS powoduje, że polecenia są automatycznie zatwierdzane, użycie trybu OCI_DEFAULT wymaga wyraźnego wywołania funkcji oci_commit() w celu zatwierdzenia transakcji lub oci_rollback(), aby ją wycofać.

Jednak warto zwrócić uwagę, że gdy polecenie jest wykonywane w trybie OCI_DEFAULT, utworzona wówczas transakcja nadal może być zatwierdzona bez wywoływania funkcji oci_commit(). W tym celu późniejsze polecenie należy wykonać w trybie OCI_COMMIT_ON_SUCCESS.

Powyższa technika może być zastosowana podczas grupowania dwóch lub większej liczby poleceń w pojedynczą transakcję. Aby zagwarantować, że cała transakcja będzie wycofana, kiedy wykonanie jednego z poleceń zakończy się niepowodzeniem lub zostaną uzyskane wyniki wskazujące na konieczność wycofania transakcji, można po prostu przerwać wykonywanie skryptu. W tym celu można wywołać na przykład funkcję trigger_error() wraz ze stałą E_USER_ERROR jako drugim parametrem, a tym samym wycofać transakcję bez potrzeby wywoływania funkcji oci_rollback().

Czytelnik może się zastanawiać, dlaczego omawiany jest niejawny sposób kończenia transakcji Oracle z poziomu PHP zamiast wyraźnego wywołania funkcji oci_commit() bądź oci_rollback(). W końcu druga z wymienionych funkcji stanowi zalecaną metodę kończenia transakcji. Ogólnie

rzecz biorąc, celem tej analizy jest umożliwienie łatwiejszego zrozumienia sposobu działania transakcji Oracle w skryptach PHP, które współdziałają z bazą danych za pomocą rozszerzenia OCI8.

W przykładzie, który zostanie omówiony poniżej, użyto struktur danych, które zdefiniowano w podrozdziale „Przykład użycia podprogramu składowanego", znajdującym się w rozdziale 3. Zanim jednak przejdziemy do przykładu, konieczne jest zmodyfikowanie wymienionych struktur danych w przedstawiony poniżej sposób. Te polecenia SQL można wykonać z poziomu narzędzia SQL*Plus po nawiązaniu połączenia z bazą danych jako `usr/usr`:

```
ALTER TABLE accounts
   ADD (num_logons INT);
UPDATE accounts
   SET num_logons = 0;
COMMIT;
DELETE logons;
ALTER TABLE logons
   ADD CONSTRAINT log_time_day
   CHECK (RTRIM(TO_CHAR(log_time, 'Day'))
      NOT IN ('Saturday', 'Sunday'));
```

Wydanie polecenia `ALTER TABLE` w powyższym bloku kodu powoduje dodanie do tabeli `accounts` kolumny `num_logons` typu `INT`. W nowej kolumnie będzie przechowywana liczba udanych operacji logowania każdego użytkownika. W tym celu gdy uwierzytelnienie użytkownika zakończy się powodzeniem, trzeba będzie zwiększyć liczbę operacji logowania przechowywaną w polu `num_logons`.

Oczywiście, nadal można się obejść bez tej kolumny, wydając względem tabeli `logons` zapytanie podobne do poniższego:

```
SELECT count(*) FROM logons WHERE usr_id='bob';
```

Jednak wraz ze wzrostem liczby operacji logowania powyższe zapytanie będzie bardzo kosztowne jak na otrzymanie informacji o liczbie logowań przeprowadzonych przez danego użytkownika.

Po dodaniu komuny `num_logons` do tabeli `accounts` nowej kolumnie należy ustawić wartość początkową wynoszącą `0`. Ewentualnie wcześniejsze polecenie `ALTER TABLE` można było wydać z użyciem klauzuli `DEFAULT` względem kolumny `num_logons`, na przykład następująco:

```
ALTER TABLE accounts
   ADD (num_logons INT DEFAULT 0);
```

W omawianym bloku kodu następuje wyraźne zatwierdzenie transakcji, aby zmiany wprowadzone przez operację `UPDATE` były trwałe.

W kolejnym kroku usuwane są wszystkie rekordy tabeli `logons`. Ten krok jest wymagany, aby zagwarantować prawidłowe wykonanie polecenia `check constraint` zdefiniowanego w następnym

kroku. W omawianym przykładzie można pominąć ten krok, jeżeli tabela logons nie zawiera rekordów utworzonych w sobotę lub niedzielę, co umożliwi prawidłowe wykonanie polecenia check constraint zdefiniowanego w kolejnym kroku. W przeciwnym razie podczas próby wykonania polecenia ALTER TABLE zostanie wyświetlony następujący komunikat błędu:

```
ERROR at line 2:
ORA-02293: cannot validate (USR.LOG_TIME_DAY) - check constraint violated
```

W skrypcie definicja check constraint obejmuje kolumnę log_time tabeli logons. Wymienione ograniczenie uniemożliwia wstawianie nowych rekordów do tabeli logons w sobotę lub niedzielę. Pozwala to na modyfikację systemu uwierzytelniania w taki sposób, aby uniemożliwić użytkownikom logowanie się w soboty i niedziele, a tym samym pozwolić na logowanie jedynie w dni robocze. W późniejszym czasie takie ograniczenie zawsze będzie można usunąć poprzez wydanie następującego polecenia:

```
ALTER TABLE logons DROP CONSTRAINT log_time_day;
```

Wróćmy jeszcze do polecenia ALTER TABLE, które przedstawiono powyżej. Warto zwrócić uwagę na użyty format 'Day', podany jako drugi parametr funkcji TO_CHAR(). Nakazuje on funkcji TO_CHAR() konwersję daty przechowywanej w polu log_time na postać dnia tygodnia. Następnie stosowany jest operator NOT IN, który powoduje wykluczenie soboty (Saturday) i niedzieli (Sunday) z listy dozwolonych dni.

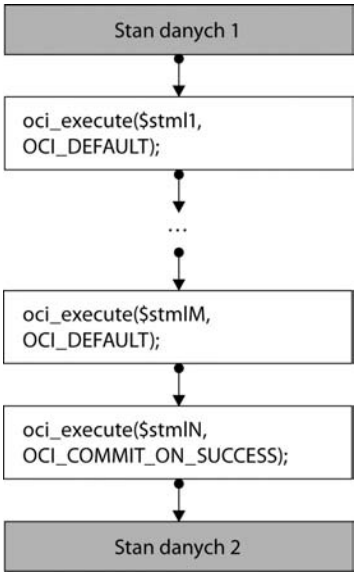
Należy pamiętać, że baza danych Oracle rozróżnia wielkość liter podczas dopasowania. Dlatego też jeśli podano 'Day' jako drugi parametr funkcji TO_CHAR(), dni tygodnia na liście znajdującej się po prawej stronie operatora NOT IN należy podać jako: 'Saturday', 'Sunday'. Miałyby one postać 'SATURDAY', 'SUNDAY', gdyby drugi parametr funkcji TO_CHAR() został podany jako 'DAY'.

Powyższe polecenia powodują przeprowadzenie wszystkich wymaganych modyfikacji struktury bazy danych. Po ich wykonaniu można więc przejść do przykładu, którego zadaniem jest zilustrowanie sposobów tworzenia transakcji poprzez wykonanie polecenia DML w trybie OCI_DEFAULT, a następnie wyraźnego zakończenia tej transakcji po wykonaniu kolejnego polecenia, ale w trybie OCI_COMMIT_ON_SUCCESS.

W rzeczywistości oczywiście może zajść potrzeba użycia więcej niż tylko dwóch poleceń w transakcji. W tym celu można wykonać w trybie OCI_DEFAULT wszystkie polecenia (poza ostatnim), które mają zostać zgrupowane w pojedynczej transakcji, a następnie wykonać ostatnie polecenie w trybie OCI_COMMIT_ON_SUCCESS — transakcja będzie zakończona.

Graficzne przedstawienie całego procesu pokazano na rysunku 4.3.

Przedstawiony poniżej skrypt prezentuje, w jaki sposób architektura pokazana na rysunku 4.3 może być zaimplementowana w PHP. Warto zauważyć, że w przeciwieństwie do funkcji login(), omówionej w podrozdziale „Przykład użycia podprogramu składowanego", który znajduje się w rozdziale 3., przedstawiona poniżej funkcja login() zatrzymuje wykonywanie i zwraca



Rysunek 4.3. Graficzne przedstawienie procesu wykonywania omówionego powyżej bloku kodu

wartość `false`, kiedy wstawienie rekordu do tabeli `logons` zakończy się niepowodzeniem. Ma to sens, ponieważ obecnie nowy rekord jest wstawiany do tabeli `logons` nie tylko w celu zapisania informacji o logowaniu, ale także sprawdzenia, czy wstawiane dane stosują się do zasad biznesowych. Wymienione zasady oznaczają, że do tabeli `logons` nie może być wstawiony żaden rekord zawierający w kolumnie `log_time` datę, dla której dniem tygodnia jest sobota bądź niedziela.

```
<?php
   // Plik: userLoginTrans.php.
   function login($usr, $pswd) {
      if(!$rsConnection = oci_connect('usr', 'usr', '//localhost/orcl')) {
         $err = oci_error();
         trigger_error('Nie można nawiązać połączenia z bazą danych: '
            . $err['message'], E_USER_ERROR);
      };
      $query = "SELECT full_name, num_logons FROM accounts
         WHERE usr_id = :userid AND pswd = :passwd";
      $stmt = oci_parse($rsConnection,$query);
      oci_bind_by_name($stmt, ':userid', $usr);
      oci_bind_by_name($stmt, ':passwd', $pswd);
      if (!oci_execute($stmt)) {
         $err = oci_error($stmt);
         trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
            . $err['message'], E_USER_ERROR);
      }
      if (!$arr = oci_fetch_array($stmt, OCI_ASSOC)) {
         print "Podano nieprawidłową nazwę użytkownika lub hasło.";
         return false;
```

```
        }
        $num_logons=$arr['NUM_LOGONS']+1;
        oci_free_statement($stmt);
        $query = "UPDATE accounts SET num_logons = num_logons + 1";
        $stmt = oci_parse($rsConnection,$query);
        if (!oci_execute($stmt, OCI_DEFAULT)) {
           $err = oci_error($stmt);
           trigger_error('Operacja uaktualnienia zakończyła się niepowodzeniem: '
              . $err['message'], E_USER_WARNING);
           return false;
        }
        oci_free_statement($stmt);
        $query = "INSERT INTO logons VALUES (:userid, SYSDATE)";
        $stmt = oci_parse($rsConnection,$query);
        oci_bind_by_name($stmt, ':userid', $usr);
        if (!oci_execute($stmt, OCI_COMMIT_ON_SUCCESS)) {
           $err = oci_error($stmt);
           trigger_error('Operacja wstawienia zakończyła się niepowodzeniem: '
              . $err['message'], E_USER_WARNING);
           if ($err['code']=='02290'){
              print "Nie można nawiązać połączenia w sobotę lub niedzielę.";
           }
           return false;
        }
        print "Witaj ".$arr['FULL_NAME']."<br/>";
        print "Odwiedziłeś nas już ".$num_logons." raz(y)";
        session_start();
        $_SESSION['user']=$usr;
        return true;
     }
?>
```

Jak już wcześniej wspomniano, kolumna `num_logons` w tabeli `accounts` przechowuje liczbę zakończonych powodzeniem operacji logowania każdego użytkownika. W skrypcie zdefiniowano polecenie `UPDATE`, zwiększające wartość pola `num_logons` w rekordzie reprezentującym użytkownika, którego dane uwierzytelniające zostały użyte podczas procesu uwierzytelniania.

Poprzez wykonanie tego polecenia w trybie `OCI_DEFAULT` następuje utworzenie nowej transakcji. Ma to sens, ponieważ wówczas istnieje możliwość wycofania zmian wprowadzonych przez operację `UPDATE`, jeśli kolejna operacja wstawiania danych do tabeli `logons` zakończy się niepowodzeniem.

Jeżeli wykonanie operacji `UPDATE` zakończy się niepowodzeniem, nastąpi opuszczenie funkcji `login()` i zwrócenie wartości `false` wywołującemu ją skryptowi. W ten sposób skrypt otrzymuje informacje, że uwierzytelnienie nie powiodło się.

Następnie w omawianym przykładzie zdefiniowane jest polecenie `INSERT`, które jest wykonywane po zakończonym powodzeniem uwierzytelnieniu użytkownika. Wykonanie go powoduje zwiększenie licznika informującego o przeprowadzonej liczbie prawidłowych operacji logowania.

Wykonanie polecenia INSERT w trybie OCI_COMMIT_ON_SUCCESS w omawianym skrypcie gwarantuje, że transakcja zostanie zatwierdzona w przypadku powodzenia operacji, a w przeciwnym razie wycofana. Oznacza to, że zmiany wprowadzone zarówno przez polecenie INSERT, jak i UPDATE albo staną się trwałe, albo będą wycofane.

Jak Czytelnik zapewne pamięta, wartością zwrotną funkcji oci_error() jest tablica asocjacyjna dwóch elementów. Pierwszy z nich — code — zawiera kod błędu Oracle, a drugi, message — komunikat błędu opisujący ten błąd. W omawianym przykładzie następuje sprawdzenie, czy kod błędu Oracle jest równy 02290. Jeżeli tak, oznacza to wystąpienie błędu związanego ze złamaniem nałożonego ograniczenia. Ponieważ w tabeli logons zdefiniowano tylko jedno ograniczenie (uniemożliwiające wstawiania nowych rekordów do tabeli logons w soboty i niedziele), można poinformować użytkownika o braku możliwości uzyskania połączenia w sobotę bądź niedzielę.

Jeżeli w omawianym skrypcie wykonanie polecenia INSERT zakończy się niepowodzeniem, nastąpi zakończenie działania funkcji login() wraz z wartością zwrotną false. Wskazuje to wywołującemu ją skryptowi, że uwierzytelnienie nie powiodło się. W przypadku uwierzytelnienia zakończonego powodzeniem można podjąć odpowiednie działania, na przykład wyświetlić komunikat powitania i utworzyć nową sesję.

Aby teraz zobaczyć w działaniu nowo utworzoną funkcję login(), można wykorzystać poniższy prosty skrypt:

```
<?php
   // Plik: testLoginTrans.php.
   require_once "userLoginTrans.php";
   if (login('bob','pswd')) {
      if (isset($_SESSION['user'])) {
         print '<p>'.'Twoja nazwa konta: '.$_SESSION['user'].'</p>';
      } else {
         print '<p>'.'Zmienna sesji przedstawiająca nazwę konta
            nie została ustawiona.'.'</p>';
      }
   }else {
      print '<p>'.'Uwierzytelnienie zakończyło się niepowodzeniem'.'</p>';
   }
?>
```

Jeżeli przedstawiony powyżej skrypt *testLoginTrans.php* zostanie uruchomiony w sobotę lub niedzielę, spowoduje wyświetlenie następującego komunikatu:

```
Nie można nawiązać połączenia w sobotę lub niedzielę.
Uwierzytelnienie zakończyło się niepowodzeniem.
```

Jednak uruchomienie skryptu w dniu roboczym spowoduje wygenerowanie następujących danych wyjściowych:

```
Witaj Bob Robinson
Odwiedziłeś nas już 1 raz(y)
Twoja nazwa konta: bob
```

Każde kolejne uruchomienie skryptu *testLoginTrans.php* w dniu roboczym spowoduje zwiększenie licznika odwiedzin strony przez użytkownika Bob Robinson. Jednak wykonanie tego skryptu w sobotę lub niedzielę nie powoduje zwiększenia wartości licznika. Powyższy test udowadnia, że wszystko działa zgodnie z założeniami.

Przenoszenie kodu transakcyjnego do bazy danych

Teraz, gdy Czytelnik dysponuje już działającym rozwiązaniem transakcyjnym zaimplementowanym przede wszystkim w PHP, warto zastanowić się, w jaki sposób można zmniejszyć ilość kodu PHP poprzez przeniesienie części logiki biznesowej aplikacji do bazy danych.

Używanie wyzwalaczy

Pracę można rozpocząć od zdefiniowania wyzwalacza `BEFORE INSERT`, obejmującego tabelę `logons`, który będzie automatycznie uaktualniał tabelę `accounts` poprzez zwiększanie wartości pola `num_logons` w odpowiednim rekordzie. W ten sposób zostanie wyeliminowana potrzeba wykonywania tej operacji `UPDATE` z poziomu kodu PHP.

Przedstawione poniżej polecenie SQL powoduje utworzenie wyzwalacza. Należy je wykonać z poziomu narzędzia SQL*Plus po nawiązaniu połączenia z bazą danych jako `usr/usr`:

```
CREATE OR REPLACE TRIGGER logons_insert
   BEFORE INSERT
   ON logons
   FOR EACH ROW
BEGIN
   UPDATE accounts
   SET num_logons = num_logons + 1
   WHERE usr_id = :new.usr_id;
END;
/
```

Po utworzeniu przedstawionego powyżej wyzwalacza `logons_insert` należy z kodu funkcji `login()`, która znajduje się w skrypcie *userLoginTrans.php*, usunąć poniższy fragment:

```
$query = "UPDATE accounts SET num_logons = num_logons + 1";
   $stmt = oci_parse($rsConnection,$query);
   if (!oci_execute($stmt, OCI_DEFAULT)) {
      $err = oci_error($stmt);
      trigger_error('Operacja uaktualnienia zakończyła się niepowodzeniem: '
         . $err['message'], E_USER_WARNING);
```

```
        return false;
    }
    oci_free_statement($stmt);
```

Warto zwrócić uwagę, że powyższa modyfikacja funkcji `login()` nie wymaga zmiany istniejącego kodu, który implementuje tę funkcję. Dlatego też w celu sprawdzenia uaktualnionej wersji funkcji `login()` nadal można wykorzystać skrypt *testLoginTrans.php*, przedstawiony w poprzedniej sekcji. Powinien on wygenerować takie same jak poprzednio dane wyjściowe.

Wycofanie na poziomie polecenia

Przeglądając kod uaktualnionej funkcji `login()`, Czytelnik może zauważyć, że nie powoduje ona wykonania jakiegokolwiek polecenia w trybie `OCI_DEFAULT`, a tym samym nie tworzy transakcji. Zamiast tego polecenie `INSERT` jest wykonywane w trybie `OCI_COMMIT_ON_SUCCESS`. Oznacza to, że każdy błąd wykryty podczas działania polecenia `INSERT` spowoduje wycofanie wszystkich zmian wprowadzonych przez polecenie `INSERT`. Z kolei jeżeli wykonanie polecenia `INSERT` zakończy się powodzeniem, zmiany zostaną automatycznie zatwierdzone.

Jak dotąd wszystko idzie dobrze. Co się jednak zdarzy, jeśli polecenie `UPDATE`, wywołane z poziomu wyzwalacza, zakończy swoje działanie niepowodzeniem? Czy spowoduje to wycofanie zmian wprowadzonych przez polecenie `INSERT`? Prostym testem pomagającym w odpowiedzi na te pytania jest tymczasowe zmodyfikowanie polecenia `UPDATE` w wyzwalaczu `logons_trigger` w taki sposób, aby działanie operacji `UPDATE` zawsze kończyło się niepowodzeniem. Następnie należy uruchomić skrypt *testLoginTrans.php*, który omówiono w podrozdziale „Nadzorowanie transakcji z poziomu PHP", i zobaczyć co się zdarzy.

W celu ponownego utworzenia wyzwalacza, aby wykonanie jego polecenia `UPDATE` zawsze kończyło się niepowodzeniem, można użyć przedstawionego poniżej polecenia SQL:

```
CREATE OR REPLACE TRIGGER logons_insert
   BEFORE INSERT
   ON logons
   FOR EACH ROW
BEGIN
   UPDATE accounts
   SET num_logons = num_logons + 'str'
   WHERE usr_id = :new.usr_id;
END;
/
```

Trzeba koniecznie zwrócić uwagę na fakt, że chociaż wykonanie polecenia `UPDATE` w wyzwalaczu zawsze będzie kończyło się niepowodzeniem, sam wyzwalacz zostanie poprawnie skompilowany.

Teraz, po uruchomieniu skryptu *testLoginTrans.php*, powinny zostać wyświetlone następujące dane wyjściowe:

```
Uwierzytelnienie zakończyło się niepowodzeniem
```

Jak można się przekonać, proces uwierzytelniania zakończył się niepowodzeniem. Aby upewnić się, że wykonanie polecenia INSERT w tabeli logons również będzie miało taki wynik, można obliczać liczbę rekordów tabeli przed i po wykonaniu skryptu *testLoginTrans.php*. Taki efekt da się osiągnąć za pomocą poniższego polecenia SQL, które wydano z poziomu narzędzia SQL*Plus po nawiązaniu połączenia z bazą danych jako usr/usr:

```
SELECT count(*) FROM logons;
```

Czytelnik powinien przekonać się, że po wykonaniu skryptu *testLoginTrans.php* liczba rekordów tabeli logons pozostanie taka sama. Stanowi to dowód, że niepowodzenie uaktualnienia tabeli logons za pomocą wyzwalacza logons_insert BEFORE INSERT, obejmującego tabelę accounts, powoduje także wycofanie zmian wprowadzonych przez polecenie INSERT.

> Ogólnie rzecz biorąc, jeżeli w trakcie wykonywania wyzwalacza nastąpi błąd, wycofane zostaną wszystkie operacje, które spowodowały uruchomienie go. Wynika to z działania tak zwanego wycofywana na poziomie rekordu — czyli każdy błąd powstały podczas wykonywania polecenia spowoduje wycofanie wszystkich zmian wprowadzonych przez to polecenie.

Jednak powyższe stwierdzenie nie zawsze jest prawdziwe. Na przykład wyzwalacz logons_insert może zostać zaimplementowany w taki sposób, że efekty działania polecenia INSERT nie zostaną wycofane, kiedy wykonanie polecenia UPDATE z tego wyzwalacza zakończy się niepowodzeniem. Warto przeanalizować przedstawioną poniżej wersję wyzwalacza logons_insert:

```
CREATE OR REPLACE TRIGGER logons_insert
   BEFORE INSERT
   ON logons
   FOR EACH ROW
BEGIN
   UPDATE accounts
   SET num_logons = num_logons + 'str'
   WHERE usr_id = :new.usr_id;
EXCEPTION
   WHEN OTHERS THEN
      NULL;
END;
/
```

Teraz, po uruchomieniu skryptu *testLoginTrans.php* powinny zostać wyświetlone dane wyjściowe o podobnej postaci:

```
Witaj Bob Robinson
Odwiedziłeś nas już 3 raz(y)
Twoja nazwa konta: bob
```

Następnie jeżeli skrypt zostanie uruchomiony ponownie, wyświetlana liczba operacji logowania pozostanie taka sama. Jednak po sprawdzeniu liczby rekordów tabeli logons, jak przedstawiono we wcześniejszej części podrozdziału, będzie można dostrzec, że kolejne wykonanie skryptu *testLoginTrans.php* spowodowało zwiększenie wartości tej liczby.

Wskazuje to, że chociaż wykonanie polecenia UPDATE z wyzwalacza zakończyło się niepowodzeniem, wykonanie polecenia INSERT zakończyło się powodzeniem. Wynika to z faktu, że omówiony powyżej wyzwalacz logons_insert powoduje ciche zignorowanie jakiegokolwiek błędu zgłaszanego podczas jego wykonywania. W sekcji WHEN OTHERS — która jest jedyną procedurą obsługi wyjątków w części wyzwalacza odpowiedzialnej za obsługę wyjątków — ustalono wartość NULL.

> W większości przypadków zastosowanie powyżej techniki nie jest zalecane, ponieważ powoduje zmianę oczekiwanego sposobu zachowania bazy danych. Rozsądne założenie jest takie, że jeśli podczas wykonywania jakiegokolwiek polecenia SQL wystąpi błąd, modyfikacje wprowadzone przez to polecenie są automatycznie wycofywane.

Dlatego też zamiast ustawiać wartość NULL w procedurze obsługi wyjątków, należy utworzyć kod, który będzie podejmował odpowiednie działania w odpowiedzi na wystąpienie błędu. Na przykład można wykorzystać procedurę RAISE_APPLICATION_ERROR w celu wygenerowania zdefiniowanego przez użytkownika błędu ORA. Znajdujący się poniżej fragment kodu pokazuje, w jaki sposób wyzwalacz logons_insert mógłby zostać zmodyfikowany, aby wywoływać RAISE_APPLICATION_ERROR z poziomu procedury obsługi błędów:

```
CREATE OR REPLACE TRIGGER logons_insert
   BEFORE INSERT
   ON logons
   FOR EACH ROW
BEGIN
   UPDATE accounts
   SET num_logons = num_logons + 'str'
   WHERE usr_id = :new.usr_id;
EXCEPTION
   WHEN OTHERS THEN
   RAISE_APPLICATION_ERROR(-20000, 'Uaktualnienie licznika nie powiodło się.');
END;
/
```

W powyższym wyzwalaczu procedura obsługi wyjątków wyraźnie wywołuję procedurę RAISE_
↪APPLICATION_ERROR w celu wygenerowania błędu ORA zdefiniowanego przez użytkownika.

> Jeżeli w pliku konfiguracyjnym *php.ini* parametr display_errors ma wartość On, omówiony w podrozdziale „Nadzorowanie transakcji z poziomu PHP" skrypt *userLoginTrans.php* spowoduje wyświetlenie komunikatu błędu wskazanego jako drugi parametr procedury RAISE_APPLICATION_ERROR.

Teraz, po uruchomieniu skryptu *testLoginTrans.php*, powinien zostać wyświetlony następujący komunikat:

```
Uwierzytelnienie zakończyło się niepowodzeniem
```

Liczba rekordów w tabeli `logons` powinna pozostać taka sama, co oznacza, że niepowodzenie operacji uaktualnienia tabeli `accounts` przez wyzwalacz powoduje nie tylko wycofanie zmian wprowadzonych przez polecenie `UPDATE`, ale również przez polecenie `INSERT`.

Przed zakończeniem pracy z tym przykładem należy się upewnić o ponownym utworzeniu wyzwalacza `logons_trigger`, aby klauzula `SET` w jego poleceniu `UPDATE` przedstawiała się następująco:

```
SET num_logons = num_logons + 1
```

Rozważania dotyczące izolacji transakcji

Kiedy transakcja modyfikuje rekord tabeli bazy danych, Oracle nakłada na niego blokadę utrzymywaną aż do chwili zatwierdzenia bądź wycofania tej transakcji. Celem takiego zachowania jest niedopuszczenie do sytuacji, w której dwie współbieżne transakcje będą modyfikowały ten sam rekord.

Bardzo ważne jest, aby w tym miejscu wspomnieć, że rekordy z nałożonymi blokadami nadal mogą być odczytywane zarówno przez transakcje uaktualniające rekordy, jak i inne transakcje. Różnica między dwiema wymienionymi transakcjami polega na tym, że transakcja nakładająca blokadę może zauważyć zmiany natychmiast po wykonaniu polecenia modyfikującego te rekordy. Natomiast inna transakcja nie może zobaczyć tych zmian aż do chwili zatwierdzenia zmian przez transakcję, która nałożyła blokadę.

Podczas gdy zastosowany w bazie danych Oracle mechanizm nakładania blokad został szczegółowo omówiony w dokumentacji Oracle (w rozdziale „Data Concurrency and Consistency" w podręczniku użytkownika *Oracle Database Concepts*), w podrozdziale przedstawiono ogólny opis działania izolacji transakcji w aplikacjach PHP/Oracle.

Którą funkcję rozszerzenia OCI8 służącą do nawiązywania połączenia należy wybrać?

Jak Czytelnik pamięta z podrozdziału „Funkcje OCI8, które służą do nawiązywania połączenia z Oracle" znajdującego się w rozdziale 2., rozszerzenie PHP OCI8 oferuje trzy różne funkcje służące do nawiązywania połączenia z bazą danych Oracle. Są to `oci_connect()`, `oci_new_connect()` oraz `oci_pconnect()`. Jedyną różnicą między tymi funkcjami jest rodzaj nawiązywanego połączenia z bazą danych.

Zarówno funkcja `oci_connect()`, jak i `oci_pconnect()` używają bufora połączenia bazy danych, tym samym eliminując koszt nawiązania połączenia w trakcie każdego żądania. Różnica między dwiema wymienionymi funkcjami polega na tym, że połączenie nawiązane przez `oci_connect()`

zostaje zwolnione po zakończeniu wykonywania skryptu, podczas gdy połączenia nawiązane przez funkcję `oci_pconnect()` są trwałe między wykonywaniem kolejnych skryptów.

W przeciwieństwie do `oci_connect()` i `oci_pconnect()` funkcja `oci_new_connect()` nie używa bufora połączeń i zawsze powoduje nawiązanie nowego połączenia. Trzeba ją stosować, gdy zachodzi potrzeba utworzenia w skrypcie dwóch lub większej liczby współbieżnych transakcji. Przedstawiony poniżej przykład prezentuje użycie funkcji `oci_new_connect()` w działaniu.

W poniższym skrypcie warto zwrócić uwagę na wykorzystanie funkcji `oci_new_connect()` w celu utworzenia współbieżnej transakcji. Chociaż do utworzenia pierwszej transakcji w skrypcie użyto funkcji `oci_connect()`, w celu nawiązania nowego połączenia zastosowano funkcję `oci_new_connect()`.

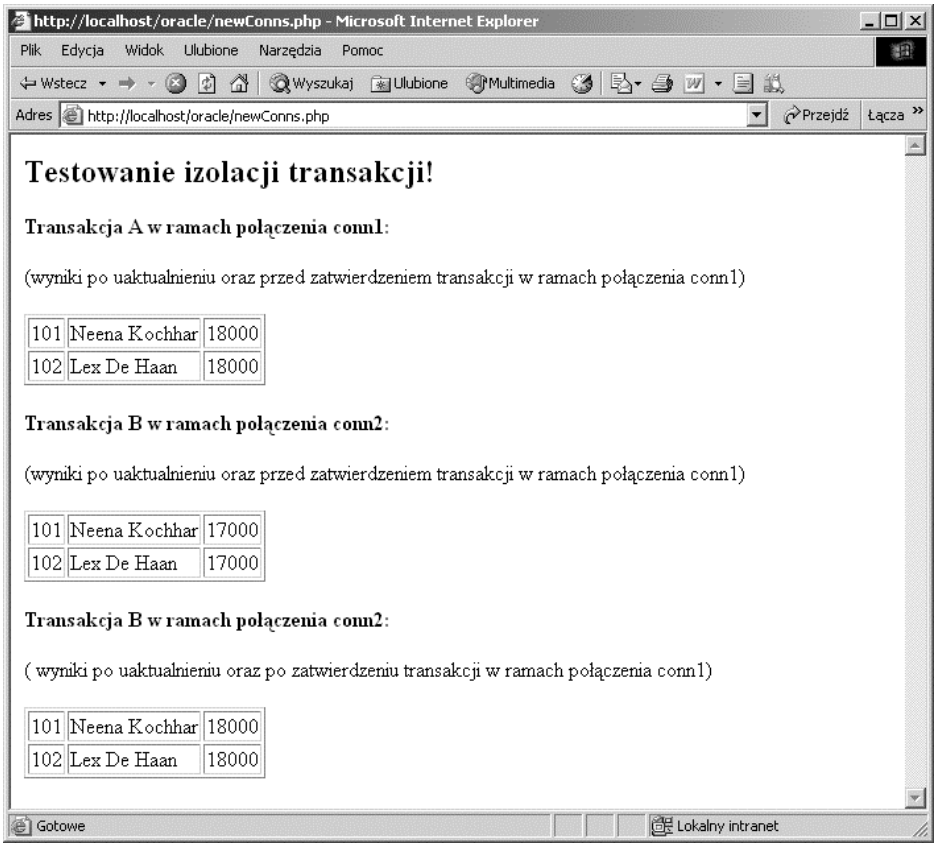
```
<?php
   // Plik: newConns.php.
   function select_emp_job ($conn, $jobno) {
      $query = "SELECT employee_id, first_name, last_name, salary
         FROM employees WHERE job_id =:jobid";
      $stmt = oci_parse($conn,$query);
      oci_bind_by_name($stmt, ':jobid', $jobno);
      if (!oci_execute($stmt, OCI_DEFAULT)) {
         $err = oci_error($stmt);
         trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
            . $err['message'], E_USER_ERROR);
      };
      print '<table border="1">';
      while ($emp = oci_fetch_array($stmt, OCI_ASSOC)) {
         print '<tr>';
         print '<td>'.$emp['EMPLOYEE_ID'].'</td>';
         print '<td>'.$emp['FIRST_NAME'].' '.$emp['LAST_NAME']. '</td>';
         print '<td>'.$emp['SALARY'].'</td>';
         print '</tr>';
      }
      print '</table>';
   }
   if(!$conn1 = oci_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   if(!$conn2 = oci_new_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   $jobno = 'AD_VP';
   $query = "UPDATE employees SET salary = 18000 WHERE job_id=:jobid";
   $stmt = oci_parse($conn1,$query);
   oci_bind_by_name($stmt, ':jobid', $jobno);
```

```
    if (!oci_execute($stmt, OCI_DEFAULT)) {
       $err = oci_error($stmt);
       trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
          . $err['message'], E_USER_ERROR);
    };
    print "<h2>Testowanie izolacji transakcji!</h2>";
    print "<h4>Transakcja A w ramach połączenia conn1:</h4>";
    print "<p>(wyniki po uaktualnieniu oraz przed zatwierdzeniem transakcji
       w ramach połączenia conn1)</p>";
    select_emp_job($conn1, $jobno);
    print "<h4>Transakcja B w ramach połączenia conn2:</h4>";
    print "<p>(wyniki po uaktualnieniu oraz przed zatwierdzeniem transakcji
       w ramach połączenia conn1)</p>";
    select_emp_job($conn2, $jobno);
    if (!oci_commit($conn1)) {
       $err = oci_error($conn1);
       trigger_error('Zatwierdzenie transakcji zakończyło się niepowodzeniem: '
          .$err['message'], E_USER_ERROR);
    }
    print "<h4>Transakcja B w ramach połączenia conn2:</h4>";
    print "<p>( wyniki po uaktualnieniu oraz po zatwierdzeniu transakcji
       w ramach połączenia conn1)</p>";
    select_emp_job($conn2, $jobno);
    $query = "UPDATE employees SET salary = 17000 WHERE job_id=:jobid";
    $stmt = oci_parse($conn1,$query);
    oci_bind_by_name($stmt, ':jobid', $jobno);
    if (!oci_execute($stmt)) {
       $err = oci_error($stmt);
       trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
          . $err['message'], E_USER_ERROR);
    };
?>
```

Na rysunku 4.4 pokazano dane wyjściowe powyższego skryptu. Jak można zobaczyć, zmiany wprowadzone przez operację `UPDATE` przeprowadzoną w transakcji działającej w ramach połączenia `conn1` z bazą danych są widoczne wewnątrz tej transakcji natychmiast po wykonaniu polecenia `UPDATE`. Nie są natomiast widoczne dla współbieżnej transakcji działającej w ramach połączenia `conn2` aż do chwili zatwierdzenia pierwszej transakcji.

Powracając do kodu źródłowego omawianego skryptu *newConns.php*: warto zwrócić uwagę, że wszystkie polecenia SQL użyte w skrypcie zostały wykonane w trybie `OCI_DEFAULT`. Gwarantuje on natychmiastowe zatwierdzenie transakcji.

Należy również zwrócić uwagę, że pierwsze połączenie w skrypcie zostało nawiązane za pomocą funkcji `oci_connect()`. Ponieważ jest to pierwsze połączenie, bufor połączeń przypisany temu skryptowi jest pusty, więc funkcja `oci_connect()` nawiąże nowe połączenie z bazą danych.



Rysunek 4.4. Testowanie izolacji transakcji

> Transakcje współbieżne trzeba tworzyć z użyciem transakcyjnie izolowanych połączeń. Podczas gdy pierwsze połączenie w skrypcie może być utworzone za pomocą funkcji `oci_connect()`, w celu utworzenia kolejnych, transakcyjnie izolowanych połączeń w skrypcie trzeba wykorzystać funkcję `oci_new_connect()`.

Następnie w celu utworzenia nowego, transakcyjnie izolowanego połączenia w skrypcie następuje wywołanie funkcji `oci_new_connect()`.

Poprzez wykonanie polecenia `UPDATE` w trybie `OCI_DEFAULT` w ramach połączenia `conn1` następuje utworzenie transakcji w tym połączeniu.

Po wykonaniu polecenia `UPDATE` efekty tej operacji będą widoczne dla innych operacji przeprowadzanych wewnątrz tej samej transakcji. Aby to udowodnić, w skrypcie wyświetlane są rekordy zmodyfikowane przez polecenie `UPDATE` wewnątrz tej samej transakcji przed jej zatwierdzeniem. Jak widać na rysunku 4.4, polecenie `SELECT` zwraca nowe wartości uaktualnionych rekordów.

Jednak podczas przeprowadzania operacji `SELECT` we współbieżnej transakcji nadal będą widoczne początkowe wartości uaktualnionych rekordów. Wynika to z faktu, że współbieżne transakcje są izolowane od zmian wprowadzonych przez niezatwierdzone transakcje. Gdy transakcja zostanie zatwierdzona, wszystkie wprowadzone przez nią zmiany staną się widoczne dla innych transakcji.

Wreszcie, za pomocą funkcji `UPDATE` następuje przywrócenie wartości początkowych uaktualnionym rekordom.

Kwestie związane z współbieżnym uaktualnianiem

Podczas projektowania aplikacji, która będzie modyfikowała dane bazy danych w środowisku wielodostępnym, programista będzie musiał zmierzyć się z dwoma wyzwaniami:

- Zapewnienie poprawności i spójności danych.
- Zagwarantowanie, że wydajność nie ucierpi z powodu nakładania blokad.

Chociaż Oracle dostarcza zestaw funkcji, które mogą pomóc osiągnąć wymienione powyżej cele, prawidłowe wykorzystanie ich jest już zadaniem programisty. Przedstawione poniżej podrozdziały koncentrują się na pewnych kwestiach związanych z współbieżnym uaktualnianiem danych, które mogą pojawić się w środowisku wielodostępnym podczas nieprawidłowego używania transakcji.

Kwestie związane z nakładaniem blokad

Jak już wcześniej wspomniano, Oracle nakłada blokadę na uaktualniany rekord, aby inne transakcje nie miały możliwości jego modyfikacji aż do zakończenia uaktualniającej go transakcji. Podczas gdy celem takiego zachowania jest zagwarantowanie poprawności danych używanych w środowisku wielodostępnym, w kiepsko zaprojektowanej aplikacji może ono spowodować powstanie znaczącego obciążenia.

Warto przyjrzeć się prostemu skryptowi pokazującemu, jak kiepsko zaprojektowany skrypt, który wykonuje długotrwające operacje, może spowodować problemy związane z nakładaniem blokad, jeśli zostanie użyty w środowisku wielodostępnym. Przedstawiony poniżej skrypt *updateSleep.php* wykonuje następujące kroki:

- Tworzy transakcję.
- Uaktualnia niektóre rekordy w tabeli `employees`.
- Wstrzymuje na 20 sekund wykonywanie skryptu.
- Wycofuje transakcję.

Wstrzymanie wykonywania poniższego skryptu za pomocą funkcji `sleep()` pozwala na symulację przeprowadzania operacji wymagającej ogromnej ilości obliczeń.

```
<?php
   // Plik: updateSleep.php.
   if(!$dbConn = oci_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   $jobno = 'AD_VP';
   $query = "
      UPDATE employees
      SET salary = salary*1.1
      WHERE job_id=:jobid";
   $stmt = oci_parse($dbConn,$query);
   oci_bind_by_name($stmt, ':jobid', $jobno);
   if (!oci_execute($stmt, OCI_DEFAULT)) {
      $err = oci_error($stmt);
      trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
         . $err['message'], E_USER_ERROR);
   };
   $updrows = oci_num_rows($stmt);
   print 'Uaktualniono '.$updrows. ' rekord(ów)'.'<br/>';
   sleep(20);
   oci_rollback($dbConn);
   print 'Transakcja została wycofana.';
?>
```

W powyższym skrypcie polecenie `UPDATE` jest wykonywane w trybie `OCI_DEFAULT`, nakazując bazie danych Oracle utworzenie transakcji.

Następnie wywołanie funkcji `sleep()` powoduje wstrzymanie wykonywania skryptu na dwadzieścia sekund, symulując tym samym przeprowadzanie operacji wymagającej ogromnej ilości obliczeń.

Wreszcie, za pomocą funkcji `oci_rollback()` następuje wycofanie transakcji. W rzeczywistości ten krok jest opcjonalny, ponieważ transakcja zostaje wycofana automatycznie, gdy skrypt kończy działanie.

Teraz uruchomiony skrypt *updateSleep.php*, który omówiono wcześniej, uaktualni wszystkie rekordy `Administration Vice President` w tabeli `employees`, zablokuje je na dwadzieścia sekund, a następnie wycofa transakcję.

Jeżeli w trakcie wymienionych dwudziestu sekund wystąpi próba uaktualnienia tych samych rekordów, ale z poziomu innego skryptu, na przykład za pomocą narzędzia SQL*Plus, będą one zablokowane, aż skrypt *updateSleep.php* zwolni blokadę nałożoną na wymienione rekordy.

Morał z powyższego skryptu jest taki, że jeśli skrypt ma przeprowadzić długotrwającą operację w bardzo obciążonym środowisku wielodostępnym, zawsze dobrym pomysłem będzie zamknięcie wszystkich aktywnych transakcji wewnątrz skryptu przed rozpoczęciem przetwarzania tej operacji.

Utracone uaktualnienia

Omówiony powyżej przykład pokazał, jak kiepsko zaprojektowana aplikacja transakcyjna może na długo zablokować zasoby bazy danych, co uniemożliwi współbieżnym transakcjom uzyskanie dostępu do tych zasobów w rozsądnym czasie. Jednak warto zwrócić uwagę, że zastosowanie podejścia bez blokad podczas modyfikowania danych bazy danych w środowisku wielodostępnym może spowodować inny problem — utracone uaktualnienia. Aby zrozumieć, na czym polega problem utraconych uaktualnień, należy przeanalizować kroki, które interaktywna aplikacja zwykle przeprowadza podczas modyfikacji informacji przechowywanych w bazie danych:

- Wybór danych z bazy danych.
- Wyświetlenie danych użytkownikowi.
- Oczekiwanie na działanie użytkownika.
- Uaktualnienie danych w bazie danych.

Powinno być całkiem oczywiste, że kiedy w powyższym przykładzie aplikacja czeka na działanie użytkownika, inny użytkownik może spowodować zmianę danych. Dlatego też jeżeli pierwszy użytkownik będzie kontynuował proces uaktualniania danych, zmiany wprowadzone przez drugiego użytkownika zostaną utracone.

Problem można lepiej zrozumieć na podstawie przykładu. Warto więc spojrzeć na skrypt *updateQuickForm.php*, który implementuje powyższe kroki za pomocą pakietu PEAR o nazwie `HTML_QuickForm`. Po uruchomieniu skryptu wykona ona następujące kroki:

- Uaktualni dwa rekordy w tabeli `employees`.
- Wygeneruje formularz proszący użytkownika o zatwierdzenie bądź wycofanie zmian.
- Zakończy działanie, wycofując wprowadzone zmiany.

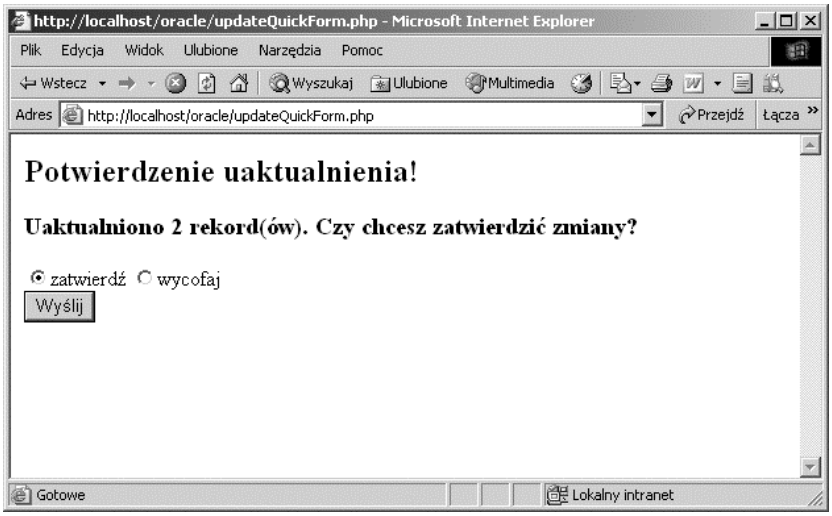
Za pomocą formularza wygenerowanego przez skrypt użytkownik może albo zatwierdzić, albo wycofać zmiany, a następnie musi kliknąć przycisk *Wyślij*. Gdy to zrobi, skrypt zostanie wywołany ponownie, tym razem wykonując poniższe kroki:

- Uaktualnienie tych samych rekordów w tabeli `employees`.
- Zatwierdzenie lub wycofanie zmian w zależności od decyzji użytkownika.
- Zakończenie działania skryptu.

Dane wyjściowe wygenerowane przez skrypt zostały pokazane na rysunku 4.5.

Jednak przed uruchomieniem skryptu *updateQuickForm.php* należy się upewnić o zainstalowaniu pakietu `PEAR::HTML_QuickForm`. Ponieważ pakiet `HTML_QuickForm` jest uzależniony od innego pakietu o nazwie `HTML_Common`, w pierwszej kolejności należy zainstalować drugi z wymienionych. Zakładając, że w systemie znajduje się zainstalowany i skonfigurowany PEAR Installer, w celu pobrania i instalacji pakietu `HTML_Common` można wydać poniższe polecenie:

```
$ pear install HTML_Common
```



Rysunek 4.5. Dane wyjściowe skryptu updateQuickForm.php wyświetlone w przeglądarce internetowej

Po instalacji pakietu `HTML_Common` można przystąpić do pobrania i instalacji pakietu `HTML_Quick` ↪`Form`. Należy wydać polecenie:

```
$ pear install HTML_QuickForm
```

Po przeprowadzeniu powyższych kroków można uruchomić skrypt *updateQuickForm.php*, którego kod źródłowy przedstawiono poniżej:

```
<?php
   // Plik: updateQuickForm.php.
   require_once 'HTML/QuickForm.php';
   if(!$dbConn = oci_connect('hr', 'hr', '//localhost/orcl')) {
      $err = oci_error();
      trigger_error('Nie można nawiązać połączenia z bazą danych: '
         . $err['message'], E_USER_ERROR);
   };
   $jobno = 'AD_VP';
   $query = "
      UPDATE employees
      SET salary = salary*1.1
      WHERE job_id=:jobid";
   $stmt = oci_parse($dbConn,$query);
   oci_bind_by_name($stmt, ':jobid', $jobno);
   if (!oci_execute($stmt, OCI_DEFAULT)) {
      $err = oci_error($stmt);
      trigger_error('Wykonanie zapytania zakończyło się niepowodzeniem: '
         .$err['message'], E_USER_ERROR);
   };
   print '<h2>Potwierdzenie uaktualnienia!</h2>';
   $updrows = oci_num_rows($stmt);
```

```
$frm=new HTML_QuickForm('frm1', 'POST');
$frm->addElement('header','msg1','Uaktualniono '.$updrows. '
   rekord(ów). Czy chcesz zatwierdzić zmiany?');
$grp[] =& HTML_QuickForm::createElement('radio', null, null,'zatwierdź', 'C');
$grp[] =& HTML_QuickForm::createElement('radio', null, null,'wycofaj', 'R');
$frm->addGroup($grp, 'trans');
$frm->setDefaults(array('trans' => 'C'));
$frm->addElement('submit','submit','Wyślij');
if(isset($_POST['submit'])) {
   if ($_POST['trans']=='C'){
      oci_commit($dbConn);
      print 'Transakcja została zatwierdzona.';
   } elseif ($_POST['trans']=='R'){
      oci_rollback($dbConn);
      print 'Transakcja została wycofana.';
   } else {
      $frm->display();
   }
} else {
   $frm->display();
}
?>
```

Przyglądając się formularzowi pokazanemu na rysunku 4.5, Czytelnik może pomyśleć, że po uaktualnieniu dwóch rekordów tabeli `employees` skrypt czeka na działanie użytkownika, pozostawiając aktywną transakcję. W rzeczywistości działa on zupełnie inaczej.

Skrypt podczas pierwszego uruchamiania faktycznie powoduje uaktualnienie tabeli `employees`, ale następnie wycofuje transakcję. W ten sposób możliwe jest zliczenie liczby rekordów modyfikowanych przez polecenie `UPDATE` i podanie tej informacji użytkownikowi. Gdy ten wybierze opcję *zatwierdź* albo *wycofaj* transakcji i kliknie przycisk *Wyślij*, skrypt ponownie wykona tę samą operację `UPDATE`. Tym razem w zależności od opcji wybranej przez użytkownika zmiany wprowadzone przez polecenie `UPDATE` będą zatwierdzone lub wycofane.

Zaletą stosowania powyższej techniki jest to, że przetwarzane rekordy nie zostają zablokowane w czasie potrzebnym na podjęcie decyzji, który przycisk kliknąć: *zatwierdź* lub *wycofaj*. Dzięki temu w tym samym czasie inne transakcje mogą przeprowadzać operacje na tych rekordach. Jednak powoduje to powstanie kolejnego problemu — utraconych uaktualnień, jak to zostało omówione we wcześniejszej części podrozdziału.

Aby uniknąć problemu utraconych uaktualnień, można zastosować strategię blokowania optymistycznego, dzięki czemu wartości uaktualnianych pól nie zostaną zmienione, kiedy użytkownik rozpocznie z nimi pracę.

Transakcje autonomiczne

Kontynuując poprzedni przykład: może wystąpić potrzeba rejestrowania prób uaktualnienia rekordów w tabeli `employees`. Aby umożliwić takie zadanie, trzeba utworzyć tabelę przechowującą sprawdzane rekordy, jak również wyzwalacz `BEFORE UPDATE`, który obejmuje tabelę `employees`. Zadaniem wyzwalacza będzie wstawianie rekordu do nowej tabeli za każdym razem, gdy ktokolwiek spróbuje uaktualnić rekord w tabeli `employees`.

W celu utworzenia wymienionej struktury danych należy wydać następujące polecenie SQL:

```
CONN usr/usr
CREATE TABLE emp_updates(
   emp_id NUMBER(6),
   job_id VARCHAR2(10),
   timedate DATE);
CONN /AS SYSDBA
GRANT INSERT on usr.emp_updates TO hr;
CONN hr/hr
CREATE OR REPLACE TRIGGER emp_updates_trigger
   BEFORE UPDATE
   ON employees
   FOR EACH ROW
BEGIN
   INSERT INTO usr.emp_updates VALUES (:new.employee_id, :new.job_id,
SYSDATE);
EXCEPTION
   WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR(-20001, 'W wyzwalaczu wystąpił błąd.');
END;
/
```

Po wykonaniu powyższego polecenia można uruchomić skrypt *updateQuickForm.php*, omówiony w poprzednim podrozdziale, i sprawdzić, czy mechanizm kontroli działa zgodnie z założeniami. W formularzu wygenerowanym przez skrypt należy zaznaczyć opcję *wycofaj*, a następnie kliknąć przycisk *Wyślij*. Teraz przy próbie wyświetlenia rekordów tabeli `emp_updates` w następujący sposób:

```
CONN usr/usr;
SELECT * FROM emp_updates;
```

Czytelnik powinien zobaczyć, że tabela `emp_updates` wciąż nie zawiera żadnych rekordów:

```
no rows selected
```

Oznacza to, że po wycofaniu operacji `UPDATE` nastąpiło wycofanie rekordu także z tabeli kontrolnej. Takie zachowanie jest oczekiwane, ponieważ nie można wycofać pewnych efektów transakcji — można albo zatwierdzić wszystkie efekty, albo wszystkie wycofać.

Próba zatwierdzenia jedynie polecenia INSERT wykonywanego wewnątrz wyzwalacza zakończy się niepowodzeniem, ponieważ w wyzwalaczu niedozwolone jest użycie żadnych poleceń nadzorujących transakcję. Dlatego też po próbie utworzenia wyzwalacza emp_updates_trigger w następujący sposób (trzeba pamiętać, aby być połączonym jako użytkownik hr — wykonując polecenie CONN hr/hr — inaczej tabela employees nie będzie widoczna i nie uda się utworzyć wyzwalacza):

```
CREATE OR REPLACE TRIGGER emp_updates_trigger
   BEFORE UPDATE
   ON employees
   FOR EACH ROW
BEGIN
   INSERT INTO usr.emp_updates VALUES (:new.employee_id,
      :new.job_id, SYSDATE);
   COMMIT;
EXCEPTION
WHEN OTHERS THEN
   ROLLBACK;
END;
/
```

uruchomienie skryptu *updateQuickForm.php* spowoduje wyświetlenie następujących komunikatów błędów:

```
Warning: oci_execute()[function.oci-execute]:ORA-04092: cannot
ROLLBACK in a trigger ORA-06512: at "HR.EMP_UPDATES_TRIGGER", line
6 ORA-04092: cannot COMMIT in a trigger ORA-04088: error during
execution of trigger 'HR.EMP_UPDATES_TRIGGER'
Fatal error: Query failed: ORA-04092: cannot ROLLBACK in a trigger
ORA-06512: at "HR.EMP_UPDATES_TRIGGER", line 6 ORA-04092: cannot
COMMIT in a trigger ORA-04088: error during execution of trigger 'HR.
EMP_UPDATES_TRIGGER
```

> Powyższe komunikaty błędów zostaną wyświetlone tylko wtedy, gdy w pliku konfiguracyjnym *php.ini* parametr display_errors ma ustawioną wartość On.

Jednym ze sposobów rozwiązania powyższego problemu jest użycie transakcji autonomicznej.

> Transakcja autonomiczna jest transakcją wewnątrz innej transakcji. Ponieważ jest całkowicie niezależna od wywołującej ją transakcji, pozwala na przeprowadzanie operacji SQL, a następnie zatwierdzanie ich lub wycofywanie bez zatwierdzania lub wycofywania wywołującej ją transakcji.

Zastosowanie transakcji autonomicznej w omawianym przykładzie pozwoli na zatwierdzenie polecenia INSERT wykonywanego przez wyzwalacz emp_updates_trigger niezależnie od transakcji

utworzonej przez skrypt *updateQuickForm.php*. Dzięki temu w tabeli `emp_updates` zostanie utworzony rekord, nawet gdy efekty operacji `UPDATE` wywołanej przez ten wyzwalacz zostaną wycofane.

Przedstawiony poniżej fragment kodu pokazuje sposób utworzenia wyzwalacza `emp_updates_trigger`, który stosuje transakcję autonomiczną.

```
CREATE OR REPLACE TRIGGER emp_updates_trigger
   BEFORE UPDATE
   ON employees
   FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO usr.emp_updates VALUES (:new.employee_id,
      :new.job_id, SYSDATE);
   COMMIT;
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
END;
/
```

Powyższy przykład prezentuje implementację transakcji autonomicznej w wyzwalaczu bazy danych. Zastosowanie tutaj transakcji autonomicznej gwarantuje, że rekord kontrolny zostanie utworzony w tabeli `emp_updates`, niezależnie od tego, czy operacja `UPDATE` względem tabeli `employees` zostanie zatwierdzona, czy wycofana.

Aby sprawdzić nowo utworzony wyzwalacz, należy ponownie uruchomić skrypt *updateQuickForm.php* i wysłać formularz wygenerowany przez skrypt po wybraniu opcji *wycofaj*. Następnie trzeba znów wyświetlić zawartość tabeli `emp_updates` (ponownie trzeba pamiętać o połączeniu na odpowiedniego użytkownika — tym razem `usr/usr`):

```
SELECT * FROM emp_updates;
```

Tym razem dane wyjściowe polecenia mogą być podobne do przedstawionych poniżej:

```
EMP_ID JOB_ID     TIMEDATE
------ ---------- ---------
   101 AD_VP      29-MAY-06
   102 AD_VP      29-MAY-06
   101 AD_VP      29-MAY-06
   102 AD_VP      29-MAY-06
```

Warto zwrócić uwagę, że chociaż próba dotyczyła uaktualnienia tylko dwóch rekordów tabeli `employees`, do tabeli kontrolnej `emp_updates` zostały wstawione cztery rekordy. Trzeba pamiętać, że w rzeczywistości skrypt *updateQuickForm.php* dwukrotnie przeprowadza operację `UPDATE`. Po raz pierwszy w celu obliczenia liczby rekordów przeznaczonych do uaktualnienia. Natomiast druga operacja faktycznie uaktualnia te rekordy.

Podsumowanie

Niektóre operacje wykonywane względem bazy danych mają sens jedynie po zgrupowaniu ich. Klasycznym przykładem jest operacja przelewu środków pieniężnych między dwoma kontami bankowymi. Jedynym sposobem bezpiecznego przeprowadzenia tego rodzaju operacji pozostaje użycie transakcji. Zastosowanie transakcji pozwala na zgrupowanie poleceń SQL w logiczne, niewidoczne jednostki pracy, z których każda może być albo w całości zatwierdzona, albo w całości wycofana.

W rozdziale Czytelnik dowiedział się, kiedy i jak wykorzystywać transakcje w aplikacjach PHP/Oracle. Analiza rozpoczęła się od ogólnego przedstawienia transakcji Oracle oraz wyjaśnienia powodów, dla których programista miałby ich używać w aplikacjach PHP zbudowanych na Oracle. Następnie omówiono organizację aplikacji PHP/Oracle w celu efektywnego kontrolowania transakcji — skoncentrowano się na korzyściach wynikających z przeniesienia logiki biznesowej aplikacji transakcyjnej z PHP do bazy danych. Czytelnik dowiedział się również, które funkcje rozszerzenia OCI8 służące do nawiązywania połączenia należy wybierać podczas używania transakcji, a także jak tworzyć współbieżne transakcje w ramach tego samego skryptu. Wreszcie — zaprezentowano wywoływanie niezależnej transakcji z wewnątrz innej transakcji oraz przedstawiono sytuacje, w których zastosowanie takiego rozwiązania może być pożądane.