

Koncepcja automatycznego tworzenia modułu auditingu w aplikacjach użytkowych

Artur Gramacki

Uniwersytet Zielonogórski

Tworząc aplikację bazodanową (zwłaszcza taką, która jest używana przez dużą liczbę użytkowników) należy zadbać o to, aby stworzyć również mechanizmy umożliwiające w miarę wszechstronne kontrolowanie i rejestrowanie tego, co dzieje się z gromadzonymi danymi (*ang. audit*). Chodzi o rejestrowanie wszelkich wykonywanych operacji DML (czyli w praktyce głównie wstawianie, modyfikowanie i kasowanie wierszy w tabelach) oraz DDL (np. tworzenie nowych obiektów).

Wykorzystywanie do tego celu istniejącego w bazie danych Oracle systemowego mechanizmu auditingu jest niewskazane, gdyż powinien on być używany w zasadzie tylko do doraźnego monitorowania stanu bazy danych w przypadkach awaryjnych, np. poszukiwania przez administratora przyczyn niewłaściwego zachowywania się systemu. Używanie go wymaga ponadto posiadania uprawnień AUDIT SYSTEM, które zwykle nie powinno być przyznawane nikomu innemu prócz administratorowi bazy.

Moduł auditingu w aplikacjach użytkowych wykorzystuje odpowiednio skonstruowane wyzwalacze bazodanowe, za pomocą których rejestrowane są wszelkie operacje wykonywane na obiektach kontrolowanego schematu. Nie jest wymagane przy tym posiadanie uprawnień DBA – wystarczą uprawnienia właściciela schematu. Ponieważ słownik bazy danych (*ang. database dictionary*) zawiera komplet informacji o wszystkich obiektach schematu, przeto możliwe jest stworzenie systemu, który automatycznie (a w praktyce prawie automatycznie) budowałby taki moduł niezależnie od złożoności danego schematu. System taki składa się z odpowiednio skonstruowanych skryptów SQL*Plus-a oraz pakietów PL/SQL. Referat pokazuje, jak taki system stworzyć i jak go używać w praktyce.

Informacja o autorze:

dr inż. Artur Gramacki – jest pracownikiem naukowym w Instytucie Informatyki i Elektroniki Uniwersytetu Zielonogórskiego. Zajmuje się projektowaniem, wykonywaniem oraz wdrażaniem aplikacji bazodanowych usprawniających szeroko rozumiane zarządzanie Uczelnią. Od wielu lat prowadzi również zajęcia dydaktyczne dotyczące projektowania baz danych, działania i administrowania systemami zarządzania bazami danych oraz wykorzystania technologii Oracle w budowie aplikacji użytkowych.

1. Wstęp

Tworząc aplikację bazodanową (zwłaszcza taką, która jest używana przez dużą liczbę użytkowników) należy zadbać o to, aby stworzyć również mechanizmy umożliwiające w miarę wszechstronne kontrolowanie i rejestrowanie tego, co dzieje się z gromadzonymi danymi (ang. *audit*). Chodzi w praktyce o rejestrowanie wszelkich wykonywanych operacji typu DML, czyli głównie wstawianie, modyfikowanie i kasowanie wierszy w tabelach.

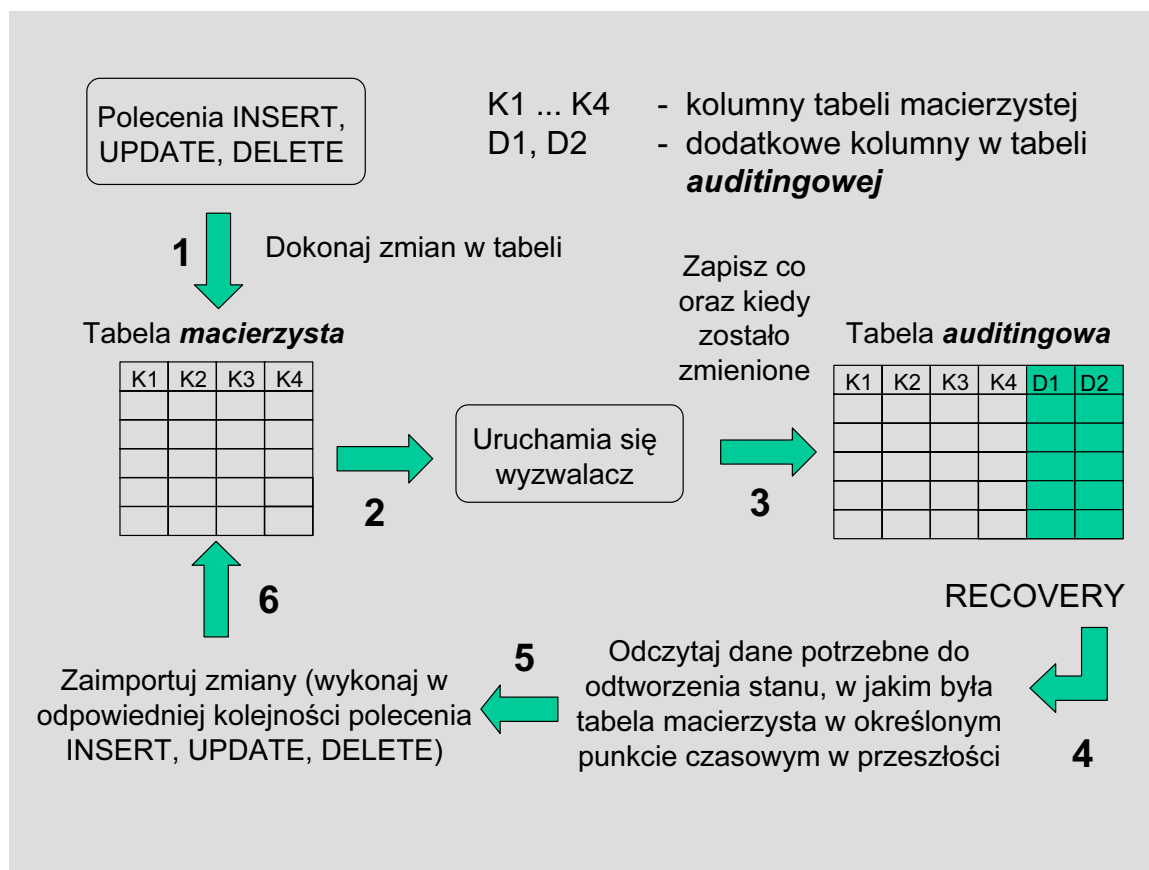
Wykorzystywanie do tego celu istniejącego w bazie danych Oracle systemowego mechanizmu auditingu na dłuższą metę nie jest niewskazane, gdyż powinien on być używany w zasadzie tylko do doraźnego monitorowania stanu bazy danych w przypadkach awaryjnych, np. poszukiwania przez administratora przyczyn niewłaściwego zachowywania się systemu. Używanie go wymaga ponadto posiadania uprawnień `AUDIT SYSTEM`, które zwykle nie powinno być przyznawane nikomu innemu prócz administratorowi bazy.

Modułu auditigu w aplikacjach użytkowych wykorzystuje odpowiednio skonstruowane (i automatycznie wygenerowane) wyzwalacze bazodanowe poziomu tabel (ang. *table-level trigger*), z pomocą których rejestrowane są wszelkie operacje wykonywane na obiektach kontrolowanego schematu. Nie jest wymagane przy tym posiadanie uprawnień DBA – wystarczą uprawnienia właściciela schematu.

Na Rysunku 1 pokazano schematycznie w jaki sposób, z użyciem wyzwalaczy bazodanowych oraz tzw. *tabel auditingowych* przechowujących dane historyczne, zrealizować wspomniany moduł auditingu. Należy również podkreślić, że przedstawiona tutaj idea wykorzystania wyzwalaczy oraz tabel auditingowych jest dość powszechnie znana. Celem artykułu jest jednak przedstawienie w jaki sposób pokazane na Rysunku 1 struktury stworzyć automatycznie (w rzeczywistości prawie automatycznie, gdyż pewne decyzje co do działania tego modułu powinien świadomie podjąć właściciel audytowanego schematu).

Dodatkowo zwrócono również uwagę na możliwość zbudowania prostego w użyciu pakietu do wykonania operacji odtworzenia do określonego punktu czasowego w przeszłości (odpowiednik opcji `UNTIL TIME` polecenia `RECOVER`). Proces odtwarzania wykorzystuje oczywiście opisane wyżej obiekty auditingowe. Pakiet ten może być używany przez właściciela danego schematu do odzyskiwania utraconych lub omyłkowo zmienionych danych bez potrzeby angażowania do tego administratora bazy danych i/lub administratora systemu operacyjnego (oczywiście mowa tutaj o sytuacjach, gdy utrata danych nie jest spowodowana awarią systemową). Ze względu na szczupłość miejsca oraz fakt, że pakiet ten jest w tej chwili intensywnie tworzony (i z pewnością jego finalna wersja będzie nieco różniła się od aktualnie istniejącej), zrezygnowano z podawania szczegółów implementacyjnych.

Ponieważ słownik bazy danych (ang. *database dictionary*) zawiera komplet informacji o wszystkich istniejących w danej chwili obiektach schematu, przeto możliwe jest stworzenie systemu, który (prawie) automatycznie budowałby taki moduł niezależnie od złożoności danego schematu. System taki składa się z odpowiednio skonstruowanych skryptów SQL*Plus-a oraz procedur i funkcji PL/SQL.



Rys. 1. Zasada rejestrowania wszelkich zmian zachodzących w tabelach z pomocą wyzwalaczy poziom tabel oraz z użyciem tabeli auditingowej do wykonania operacji odtworzenia (ang. *recovery*) danych do pewnego punktu w przeszłości.

Użytkownik wprowadza nowe lub modyfikuje istniejące dane (strzałka nr 1). Powoduje to uruchomienie odpowiednio skonstruowanych wyzwalaczy poziom tabel (strzałka nr 2), z pomocą których wszelkie dokonywane zmiany zostają zarejestrowane w tabelach auditingowych (strzałka nr 3). Tak zarejestrowane dane mogą być następnie użyte do wykonania operacji odtworzenia (strzałka nr 4). Po przygotowaniu odpowiednich skryptów zawierających polecenia INSERT, UPDATE oraz DELETE (strzałka nr 5) można je wykonać na tabeli macierzystej i tym samym odtworzyć stan tej tabeli do określonego punktu w przeszłości (strzałka nr 6).

2. Generatory kodu SQL oraz PL/SQL

Chcąc zbudować system, który automatycznie wygeneruje pewien kod w języku SQL lub PL/SQL najłatwiej jest wykorzystać do tego celu właśnie języki SQL, PL/SQL oraz informacje przechowywane w odpowiednich tabelach słownika bazy danych. Przykładowo, aby usunąć wszystkie tabele ze schematu użytkownika należy po pierwsze odnaleźć nazwy wszystkich występujących w tym schemacie tabel a następnie wydać odpowiednią ilość razy polecenie DROP TABLE. Czynność ta, przy dużej liczbie istniejących w schemacie użytkownika tabel, jest dość żmudnym zadaniem. Pracę można jednak znacznie uprościć pisząc następujące polecenie w języku SQL:

```
SELECT 'DROP TABLE '||table_name||' CASCADE CONSTRAINTS;' FROM USER_TABLES;
```

Wykonanie powyższego polecenia SELECT spowoduje wyświetlenie na konsoli tylu wierszy, ile jest tabel w bieżącym schemacie użytkownika. Każdy wiersz zawiera polecenie kasowania jednej tabeli. Chcąc otrzymać bardziej funkcjonalny i gotowy do uruchomienia skrypt, należy

powyższe polecenie „obudować” dodatkowo odpowiednimi komendami programu SQL*Plus, przykładowo:

```
SET ECHO OFF
SET TERM OFF
SET FEEDBACK OFF
SET PAGESIZE 0
SPOOL drop_all_tables.sql
SELECT 'DROP TABLE '||table_name||' CASCADE CONSTRAINTS;' FROM USER_TABLES;
SPOOL OFF
/
-- Wykonaj utworzony skrypt
SET FEEDBACK ON
SET TERM ON
SET ECHO ON
@drop_all_tables.sql
```

Skrypt ten należy wywołać wprost z poziomu programu SQL*Plus. W bardziej złożonych przypadkach wygodniej jest użyć języka PL/SQL. Wówczas naturalnym rozwiązaniem jest skorzystanie z funkcji `dbms_output.put_line` do generowania kodu, który po wyświetleniu na ekranie należy skopiować i wstawić do pliku a następnie plik ten wykonać jako skrypt, np. w programie SQL*Plus. Można również wykorzystać do tego celu pakiet `utl_file` celem generowania kodu wprost do pliku tekstowego (z pominięciem wyświetlania go na ekranie i potrzeby ręcznego kopiowania do pliku). Kody odpowiednich procedur będą wyglądały następująco:

```
CREATE OR REPLACE PROCEDURE drop_all_tables AS
BEGIN
  FOR kursor IN (
    SELECT 'DROP TABLE '||table_name||' CASCADE CONSTRAINTS;' tekst
    FROM user_tables)
  LOOP
    DBMS_OUTPUT.PUT_LINE(kursor.tekst);
  END LOOP;
END;
/
```

oraz

```
CREATE OR REPLACE PROCEDURE drop_all_tables AS
file_handle UTL_FILE.FILE_TYPE;
BEGIN
  -- Uwaga. Aby można było utworzyć plik w katalogu c:\temp należy w pliku
  -- z parametrami bazy (INIT.ORA) umieścić wpis: UTL_FILE_DIR = c:\temp
  file_handle := UTL_FILE.FOPEN('c:\temp', 'drop_all_tables.sql', 'w');
  FOR kursor IN (
    SELECT 'DROP TABLE '||table_name||' CASCADE CONSTRAINTS;' tekst
    FROM user_tables)
  LOOP
    UTL_FILE.PUTF(file_handle, kursor.tekst||'\n');
  END LOOP;
  UTL_FILE.FCLOSE(file_handle);
END;
/
```

W analogiczny sposób można budować nawet bardzo złożone generatory.

Na kolejnym przykładzie pokazano procedurę PL/SQL, która tworzy skrypt wyświetlający szczegóły budowy tabeli podanej jako parametr wywoływanej procedury (jest to *de facto* dość dokładny odpowiednik polecenia `desc`).

```
CREATE OR REPLACE PROCEDURE desc_table (in_tab_name VARCHAR2) AS
  header VARCHAR2(2000);
  header2 VARCHAR2(2000);
BEGIN
  -- Nagłówek
  SELECT
    RPAD('Column',30)||RPAD('Null?',7)      ||RPAD('Type',20)||
    RPAD('Length',8) ||RPAD('Precision',11)||RPAD('Scale',7)
  INTO header FROM DUAL;
  SELECT
    RPAD('-',29,'-')||' '||RPAD('-',6,'-')||' '||
    RPAD('-',19,'-')||' '||RPAD('-',7,'-')||' '||
    RPAD('-',10,'-')||' '||RPAD('-',6,'-') INTO header2 FROM DUAL;
  DBMS_OUTPUT.PUT_LINE(header);
  DBMS_OUTPUT.PUT_LINE(header2);
  -- Kolejne kolumny
  FOR kursor IN (
    SELECT * FROM user_tab_columns WHERE table_name = UPPER(in_tab_name)
    ORDER BY column_ID)
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      RPAD(kursor.column_name,30) ||RPAD(kursor.nullable,7) ||
      RPAD(kursor.data_type,20) ||RPAD(kursor.data_length,8) ||
      RPAD(kursor.data_precision,11)||RPAD(kursor.data_scale,7) );
  END LOOP;
END;
/
```

Widać więc, że możliwości budowy generatorów kodu są w zasadzie nieograniczone. Wymagana jest tylko znajomość języków SQL, PL/SQL oraz szczegółów budowy odpowiednich tabel słownika bazy danych.

3. Automatyzacja procesu budowania struktur do śledzenia zmian zachodzących w bazie danych

3.1. Uwagi wstępne

Wszystkie niezbędne informacje potrzebne do zbudowania omawianego tu modułu auditingu odnajdziemy w słowniku bazy danych, patrz Rysunek 2. Uwzględniając główne założenia podane w rozdziale 1 (oraz schematycznie zobrazowane na Rysunku 1) należy utworzyć następujące elementy:

- tabele auditingowe,
- wyzwalacze dla każdej z tabel macierzystych,
- pakiety implementujące funkcje do odtwarzania danych,
- ewentualne inne pomocnicze programy i skrypty.

Nazwa tabeli	Opis
USER_TABLES	informacje o fizycznych parametrach składowania tabel
USER_TAB_COLUMNS	informacje o wszystkich kolumnach w tabelach
USER_CONSTRAINTS	informacje o wszystkich ograniczeniach zdefiniowanych na tabelach
USER_CONS_COLUMNS	informacje o wszystkich ograniczeniach zdefiniowanych na tabelach
USER_TRIGGERS	informacje o wszystkich wyzwalaczach w schemacie użytkownika

Rys. 2. Używane tabele słownika bazy danych

3.2. Specyfikacja tabel auditingowych

Podstawowe założenia, według których tworzone są tabele auditingowe są następujące:

- Dla każdej tabeli znajdującej się w schemacie użytkownika tworzymy jej „lustrzany” odpowiednik. Nadajemy jej nazwę według schematu: *nazwa_tabeli_macierzystej_AUD*. Gdy nazwa tabeli macierzystej składa się z więcej niż 26 znaków skracamy ją do 26 znaków. Gdy tak utworzona nazwa dubluje się z już istniejącą nazwą tabeli, skracamy ją do 25 znaków i dodajemy końcówkę *AUD2*, itd.
- Tabela lustrzana posiada wszystkie kolumny tabeli macierzystej, ułożone w tej samej kolejności. Dodatkowo dodajemy do każdej takiej tabeli kolumny „auditingowe” według specyfikacji pokazanej na Rysunku 3.

Nazwa kolumny	Uwagi
ID_AUD	kolumna z ograniczeniem PRIMARY KEY, typ NUMBER.
ID_AUD_2	kolumna z ograniczeniem NULL, typ NUMBER. Używana w przypadku rejestrowania operacji UPDATE. Przepisujemy do niej wartość z kolumny ID_AUD. Patrz przykłady w dalszej części artykułu.
TYP_AUD	kolumna z ograniczeniem NOT NULL oraz z ograniczeniem CHECK IN ('I', 'U', 'D') – od: INSERT, UPDATE, DELETE.
OSOBA_AUD	kolumny z ograniczeniem NOT NULL, typ VARCHAR2. W kolumnie tej wpisywać będziemy nazwę użytkownika, który dokonał operacji INSERT, UPDATE lub DELETE na danym rekordzie tabeli macierzystej.
DATA_AUD	kolumny z ograniczeniem NOT NULL, typ DATE. W kolumnie tej wpisywać będziemy datę i czas wykonania danej operacji na danym rekordzie tabeli macierzystej.

Rys. 3. Dodatkowe kolumny dodawane do tabel auditingowych

- W tabelach auditingowych odtwarzamy ograniczenia NOT NULL oraz CHECK. Ograniczeń UNIQUE oraz PRIMARY KEY nie można odtwarzać, gdyż w czasie tworzenia rekordów w tabelach auditingowych kolumny te nie będą w ogólności posiadały wartości unikalnych. Gdy nie odtwarzamy ograniczeń PRIMARY KEY, nie ma oczywiście sensu odtwarzanie ograniczeń FOREIGN KEY. Nazwy nowych ograniczeń tworzymy według schematu *nazwa_macierzystego_ograniczenia_AUD* (pod warunkiem, że są to nazwy zdefiniowane przez użytkownika. Gdy tak nie jest pozostawiamy nazwy systemowe). Gdy nazwa macierzystego ograniczenia składa się z więcej niż 26 znaków, skracamy ją do 26 znaków. Gdy tak utworzona nazwa dubluje się z już istniejącą nazwą ograniczenia skracamy ją do 25 znaków i dodajemy końcówkę *AUD2*, itd.

Poniżej pokazano procedurę, która generuje kod SQL tworzący tabelę auditingową dla podanej jako parametr tabeli macierzystej. Ze względu na wielkość oryginalnej procedury przedstawiono jej wersję uproszczoną. Uproszczenia są następujące:

- nie jest zaimplementowana obsługa fizycznych parametrów składowania,
- rozpoznawane są wyłącznie trzy typy danych (NUMBER, VARCHAR2, DATE),
- uwzględniane są wyłącznie ograniczenia NOT NULL,
- ograniczenia CHECK są ignorowane,
- ponadto tworzone ograniczenia NOT NULL mają zawsze nazwy systemowe, nawet gdy odpowiadające im ograniczenia w tabelach macierzystych mają nazwy zdefiniowane przez użytkownika.

```

CREATE OR REPLACE PROCEDURE Create_audit_table (in_tab_name VARCHAR2) AS
CURSOR cur_UsrTabCols IS
SELECT * FROM user_tab_columns WHERE table_name = UPPER(in_tab_name) ORDER BY co-
lumn_id;
-- Zmienne ( przedrostek uv - user variable )
uv_utc      cur_UsrTabCols%ROWTYPE;
uv_text     VARCHAR2(30);
uv_null     VARCHAR2(8);
uv_ile_kol  PLS_INTEGER;
uv_tab_name VARCHAR2(30) := UPPER(in_tab_name);
BEGIN
OPEN cur_UsrTabCols;

-- Sprawdzamy ile jest kolumn w tabeli.
SELECT MAX(column_id) INTO uv_ile_kol FROM user_tab_columns
WHERE table_name = uv_tab_name;

-- Obcinamy nazwę tabeli do 26 znaków.
uv_tab_name := LOWER(SUBSTR(uv_tab_name,1,26));
dbms_output.put_line('DROP TABLE '||uv_tab_name||'_AUD;'||CHR(10));
dbms_output.put_line('CREATE TABLE '||uv_tab_name||'_AUD ( ');

-- "Chodzimy" po nazwach kolumn w tabeli 'user_tab_columns'.
LOOP
  FETCH cur_UsrTabCols INTO uv_utc;
  EXIT WHEN cur_UsrTabCols%NOTFOUND;

  -- Sprawdzamy, czy dana kolumna jest NULL czy NOT NULL.
  IF uv_utc.nullable = 'N' THEN
    uv_null := 'NOT NULL';
  ELSE
    uv_null := 'NULL';
  END IF;

  -- Określamy typ danych kolumny
  IF uv_utc.data_type = 'NUMBER' THEN
    IF uv_utc.data_precision IS NULL AND uv_utc.data_scale IS NULL THEN
      uv_text := NULL;
    ELSIF uv_utc.data_scale = 0 THEN
      uv_text := '('||uv_utc.data_precision|| ')';
    ELSE

```

```

        uv_text := '('||uv_utc.data_precision||','||uv_utc.data_scale||)';
    END IF;
ELSIF uv_utc.data_type = 'DATE' THEN
    uv_text := NULL;
ELSIF uv_utc.data_type IN ('VARCHAR2', 'CHAR') THEN
    uv_text := '('||uv_utc.data_length||)';
ELSE
    raise_application_error(-20002,'Nie obsługiwany typ danych.');
```

END IF;

-- Używamy RPAD do ładnego wyrównania tekstu w pionie.

```

dbms_output.put_line(
    RPAD(LOWER(uv_utc.column_name),30) ||
    RPAD(uv_utc.data_type||uv_text,15)||RPAD(uv_null,8)||','');
END LOOP;
CLOSE cur_UsrTabCols;

-- Dodajemy do tabeli kolumny auditingowe
dbms_output.put_line(RPAD('id_aud', 30)||RPAD('NUMBER(10)',15)||'PRIMARY KEY,');
dbms_output.put_line(RPAD('id_aud_2', 30)||RPAD('NUMBER(10)',15)||'NULL ,');
dbms_output.put_line(RPAD('typ_aud', 30)||RPAD('CHAR(1)', 15)||'NOT NULL CHECK
(typ_aud IN (''I'', ''U'', ''D'')),');
dbms_output.put_line(RPAD('osoba_aud',30)||RPAD('VARCHAR2(30)',15)||'NOT NULL,');
dbms_output.put_line(RPAD('data_aud' ,30)||RPAD('DATE',15)||'NOT NULL');
```

-- Doklejamy nawias zamykający

```

dbms_output.put_line(')||CHR(10));

EXCEPTION
    WHEN OTHERS THEN
        raise_application_error(-20001, SQLERRM);
END;
/
```

Aby łatwo można było wykorzystać powyższą procedurę, należy napisać jeszcze kolejną, która będzie ją w pętli wywoływała dla wszystkich tabel macierzystych występujących w schemacie użytkownika. Wygląda ona następująco:

```

CREATE OR REPLACE PROCEDURE Create_all_audit_tables IS
CURSOR cur_tab IS SELECT table_name FROM user_tables;
uv_tab cur_tab%ROWTYPE;
BEGIN
OPEN cur_tab;
LOOP
    FETCH cur_tab INTO uv_tab;
    EXIT WHEN cur_tab %NOTFOUND;
    create_audit_table(uv_tab.table_name);
END LOOP;
CLOSE cur_tab;
END;
/
```

3.3. Specyfikacja wyzwalaczy obsługujących tabele auditingowe

Dla każdej tabeli znajdującej się w schemacie użytkownika tworzymy wyzwalacz typu BEFORE INSERT OR UPDATE OR DELETE, którego zadaniem będzie rejestrowanie wszystkich zmian dokonywanych na audytowanych tabelach. Nadajemy im nazwy według schematu: *nazwa_tabeli_macierzystej_TRG*. Gdy nazwa tabeli macierzystej składa się z więcej niż 26 znaków skracamy ją do 26 znaków. Gdy tak utworzona nazwa wyzwalacza dubluje się z nazwą już istniejącego wyzwalacza nazwę tabeli macierzystej skracamy ją do 25 znaków i dodajemy końcówkę TRG2, itd. Wszystkie wyzwalacze mają analogiczną strukturę do tej pokazanej niżej utworzonej dla przykładowej tabeli TAB (patrz Rysunek 4 w dalszej części artykułu).

```
CREATE OR REPLACE TRIGGER tab_TRG
BEFORE INSERT OR UPDATE OR DELETE ON tab
REFERENCING NEW AS new OLD AS old FOR EACH ROW
DECLARE
num NUMBER;
BEGIN
IF INSERTING THEN
INSERT INTO tab_aud VALUES
(new.id, new.kol, aud_seq.nextval, NULL, 'I', user, sysdate);
ELSIF UPDATING THEN
SELECT aud_seq.nextval INTO num FROM DUAL;
INSERT INTO tab_aud VALUES
(old.id, old.kol, num, NULL, 'U', user, sysdate);
INSERT INTO tab_aud VALUES
(new.id, new.kol, aud_seq.nextval, num, 'U', user, sysdate);
ELSIF DELETING THEN
INSERT INTO tab_aud VALUES
(old.id, old.kol, num, NULL, 'D', user, sysdate);
END IF;
END;
/
```

Odpowiedni generator tworzący powyższy wyzwalacz będzie wyglądał następująco:

```
CREATE OR REPLACE PROCEDURE Create_audit_triggers (in_tab_name VARCHAR2) IS
CURSOR cur_UsrTabCols IS SELECT * FROM user_tab_columns
WHERE table_name = UPPER(in_tab_name) ORDER BY column_ID;
uv_utc cur_UsrTabCols%ROWTYPE;
uv_tab_name VARCHAR2(30) := in_tab_name;

BEGIN
-- Obcinamy nazwę tabeli do 26 znaków.
uv_tab_name := LOWER(SUBSTR(uv_tab_name,1,26));

dbms_output.put_line('DROP TRIGGER '||uv_tab_name||'_TRG'||';' ||CHR(10));
dbms_output.put_line('CREATE OR REPLACE TRIGGER '||uv_tab_name||'_TRG');
dbms_output.put_line('BEFORE INSERT OR DELETE OR UPDATE ON '||in_tab_name);
dbms_output.put_line('REFERENCING NEW AS new OLD AS old FOR EACH ROW');
dbms_output.put_line('BEGIN');

dbms_output.put_line('IF INSERTING THEN');
dbms_output.put_line(' INSERT INTO '|| uv_tab_name ||'_AUD VALUES(');
OPEN cur_UsrTabCols;
LOOP
FETCH cur_UsrTabCols INTO uv_utc;
```

```

EXIT WHEN cur_UsrTabCols%NOTFOUND;
dbms_output.put_line(' :new.'||uv_utc.column_name||',');
END LOOP;
CLOSE cur_UsrTabCols;

-- Dodaje dodatkowe kolumny
dbms_output.put_line(' aud_seq.NEXTVAL, NULL, ''I'', user, sysdate);');

dbms_output.put_line('ELSIF DELETING THEN');
dbms_output.put_line(' NULL;');
-- Dopisać analogicznie jak dla IF INSERTING

dbms_output.put_line('ELSIF UPDATING THEN');
dbms_output.put_line(' NULL;');
-- Dopisać analogicznie jak dla IF INSERTING

dbms_output.put_line('END IF;');
dbms_output.put_line('END;'||CHR(10));

EXCEPTION
WHEN OTHERS THEN
    raise_application_error(-20001, SQLERRM);
END;
/

```

Dodatkowo, po utworzeniu wyzwalaczy, za pomocą prostego generatora korzystającego z tabeli słownikowej USER_TRIGGERS, tworzona jest procedura do szybkiego włączania / wyłączania (ENABLE / DISABLE) utworzonych wyzwalaczy. Wygląda ona następująco:

```

CREATE OR REPLACE PROCEDURE triggers_enable_disable (in_option VARCHAR2) IS
uv_option VARCHAR2(10);
BEGIN
    IF UPPER(in_option) = 'E' THEN
        uv_option := ' ENABLE';
    ELSIF UPPER(in_option) = 'D' THEN
        uv_option := ' DISABLE';
    ELSE
        raise_application_error(-20001,'Błędny parametr wejściowy.');

```

3.4. Przykład pokazujący zasadę działania tabel auditingowych

Poniżej pokazano przykładową bardzo prostą tabelę demonstracyjną (patrz Rysunek 4). Pokazano również zawartość tabeli auditingowej po wykonaniu czterech operacji UPDATE, jednej operacji INSERT oraz dwóch operacji DELETE (Rysunek 5), jak również stan tabeli demonstracyjnej po wykonaniu tych operacji (Rysunek 6). Uwaga: w przypadku operacji UPDATE w tabeli auditin-

gowej tworzone są dwa rekordy – jeden przechowuje stan przed zmianą a drugi po dokonanej zmianie. Dzięki temu bardzo łatwo jest utworzyć polecenie `UPDATE nazwa_tabeli SET ...`, które wycofa dokonana zmianę i uaktualni właściwy rekord. W przypadku operacji `DELETE` do tabeli auditingowej wpisywane są wartości poszczególnych kolumn przed dokonaniem operacji. Oczywiście dla operacji `INSERT` takie rozróżnienie jest bez znaczenia.

Na Rysunku 7 zestawiono wszystkie operacje konieczne do wykonania odtworzenia (ang. *recovery*) stanu tabeli do stanu, w jakim była w dniu 09-09-2003. Odtwarzanie oczywiście wykonujemy w kolejności odwrotnej, do tej, w jakiej wprowadzane były zmiany.

ID	KOL
100	a
101	b
102	c

Rys. 4. Przykładowa tabela TAB z danymi. Stan początkowy

ID	KOL	ID_AUD	ID_AUD_2	TYP_AUD	OSOBA_AUD	DATA_AUD
100	a	1001		U	AGRAM	09-09-2003
100	aa	1002	1001	U	AGRAM	09-09-2003
100	aa	1003		U	AGRAM	12-10-2003
100	aaa	1004	1003	U	AGRAM	12-10-2003
100	aaa	1005		U	AGRAM	15-10-2003
100	aaaa	1006	1005	U	AGRAM	15-10-2003
103	d	1007		I	AGRAM	12-11-2003
101	b	1008		U	AGRAM	25-11-2003
101	bb	1009	1008	U	AGRAM	25-11-2003
101	bb	1010		D	JGRAM	29-11-2003
100	aaaa	1011		D	JGRAM	01-12-2003

Rys. 5. Zawartość tabeli auditingowej po wykonaniu czterech operacji `UPDATE` jednej operacji `INSERT` oraz dwóch operacji `DELETE`.

ID	KOL
102	c
103	d

Rys. 6. Przykładowa tabela z danymi. Stan po wykonaniu operacji wyszczególnionych w tabeli auditingowej

TYP_AUD	Operacja do wykonania
D	<code>INSERT INTO tab VALUES (100, 'aaaa');</code>
D	<code>INSERT INTO tab VALUES (101, 'bb');</code>
U	<code>UPDATE tab SET id=101, kol='b' WHERE id='101' AND kol='bb';</code>
I	<code>DELETE FROM tab WHERE id='103' AND kol='d';</code>
U	<code>UPDATE tab SET id=100, kol='aaa' WHERE id='100' AND kol='aaaa';</code>

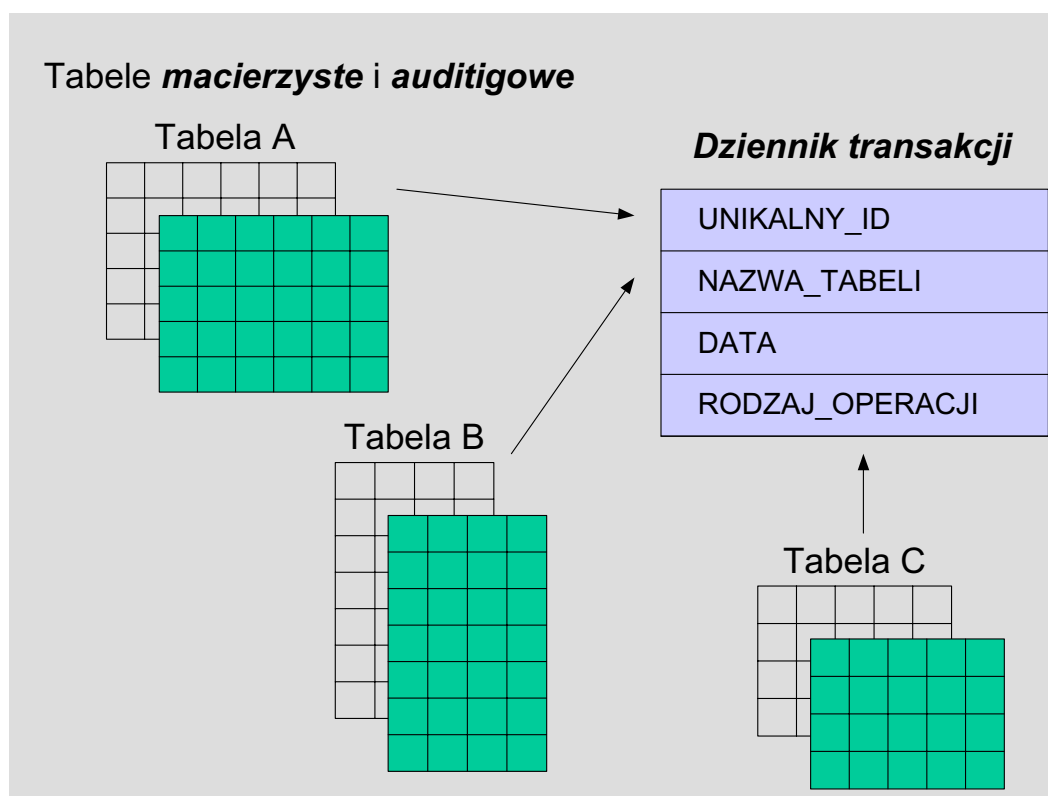
TYP_AUD	Operacja do wykonania
U	UPDATE tab SET id=100, kol='aa' WHERE id='100' AND kol='aaa';
U	UPDATE tab SET id=100, kol='a' WHERE id='100' AND kol='aa';

Rys. 7. Operacje, które należy wykonać aby powrócić do stanu pokazanego Rysunku 4

Uwaga: w rozważanym przykładzie dla uproszczenia zakładamy, że w schemacie użytkownika mamy tylko jedną tabelę. Oczywiście w rzeczywistych aplikacjach praktycznie taki przypadek nigdy nie zachodzi – tabel jest zwykle kilkadziesiąt lub więcej. Są one ponadto powiązane ze sobą za pomocą kluczy obcych i zaprezentowane poniżej podejście musi zostać rozszerzone o możliwość analizowania zmian we wszystkich tabelach przy uwzględnieniu chronologii tych zmian.

Innymi słowy, musimy traktować wszystkie tabele jako powiązana ze sobą całość i przyjmując założenie, że odtwarzamy stan całego schematu do określonego punktu w przeszłości a nie stan pojedynczej, izolowanej od reszty, tabeli. Istnieje oczywiście możliwość odtwarzania stanu zawartości tylko jednej wybranej tabeli, ale wówczas trzeba sprawdzić i ew. odtworzyć zawartość wszystkich tych tabel, które są powiązane z odtwarzaną tabelą i ich zawartość uległa „w międzyczasie” również zmianie. Podejście takie jest zaprezentowane w pracy [1].

Wymagana jest wówczas dodatkowa tabela, zwana *dziennikiem transakcji*, gdzie zapisywane są, w kolejności ich wystąpienia, wszelkie czynności wykonywane na tabelach macierzystych. Informacje zapisywane wyłącznie w tabelach auditingowych nie pozwalają na precyzyjne określenie kolejności w jakiej następowały wpisy w poszczególnych tabelach auditingowych. Rysunek 8 pokazuje zasadę działania dziennika transakcji. Kolejność dokonywanych wpisów w tabelach auditingowych określamy za pomocą wartości kolumny UNIKALNY_ID. Ponieważ poszczególne komórki tej kolumny są wypełniane liczbami podsyłanymi przez dedykowaną tylko dla tej tabeli sekwencję, więc można przyjąć, że posortowany wg. tej kolumny dziennik transakcji zawiera wpisy w kolejności chronologicznej.



Rys. 8. Zasada działania dziennika transakcji

4. Podsumowanie

W artykule przedstawiono koncepcję tzw. modułu auditingu, który wspomaga proces tworzenia tabel oraz wyzwalaczy auditingowych. Zadaniem tabel auditingowych jest przechowywanie wszelkich zmian zachodzących na tabelach macierzystych (wstawianie, modyfikacja i kasowanie rekordów). Z kolei zadaniem wyzwalaczy auditingowych jest automatyczne rejestrowanie tych zmian we wspomnianych tabelach auditingowych. Rejestrowanie zmian odbywa się z poziomu właściciela danego schematu i aby uaktywnić ten mechanizm nie są do tego potrzebne uprawnienia administratora bazy danych. Można więc powiedzieć, że omówiony tu moduł auditingu znacząco uzupełnia istniejący w bazie danych ORACLE auditing systemowy (polecenia systemowe `audit` oraz `noaudit`).

Pokazano w jaki sposób, z użyciem odpowiednio skonstruowanych skryptów SQL*Plus oraz programów PL/SQL (tzw. generatory kodu) wspomniane wyżej obiekty auditingowe utworzyć. Czynność ta jest wykonywana całkowicie automatycznie, niezależnie od wielkości i złożoności danego schematu użytkownika.

Zaprezentowane w artykule przykładowe programy należy traktować jako przykłady demonstracyjne a nie całkowicie gotowe do użycia narzędzia. Uwzględnienie bowiem wszystkich wymaganych elementów powoduje, że kody wynikowe stają się bardzo obszerne. Przykładowo zupełnie pominięto kwestię obsługi ew. błędów, która w kontekście automatycznie budowanych poleceń CREATE, INSERT, UPDATE oraz DELETE jest bardzo istotna.

Zwrócono również uwagę, na naturalna możliwość użycia danych z tabel auditingowych do zaimplementowania modułu automatycznego odtwarzania stanu zawartości tabel do pewnego momentu w przeszłości. Należy podkreślić, że chodzi tutaj o mechanizm możliwy do uruchomienia z poziomu właściciela danego schematu, bez potrzeby wdrażania systemowego mechanizmu *backup and recovery*, wymagającego oczywiście posiadania uprawnień DBA. Odtwarzanie danych na tym poziomie można więc uznać za uzupełnienie odtwarzania systemowego.

Implementacja odpowiednich skryptów SQL*Plus i programów PL/SQL nie jest w chwili obecnej całkowicie ukończona (szczególnie dotyczy to modułu automatycznego odtwarzania). Gotowe już moduły są testowane a inne ciągle powstają.

Bibliografia

1. Robson P. G.: Table Backup and Recovery using SQL*Plus, VIII Konferencja Użytkowników i Deweloperów ORACLE : Systemy informatyczne. Projektowanie, implementowanie, eksploatawanie. Zakopane, Polska, 2002 s. 9—23
2. Dokumentacja systemu ORACLE: Oracle 8i SQL Reference
3. Dokumentacja systemu ORACLE: Oracle8i Reference
4. Dokumentacja systemu ORACLE: PL/SQL User's Guide and Reference