

ORACLE®



Oracle Database 12c i SQL

Programowanie

Jason Price

Helion

Oracle
Press™

Spis treści

Wprowadzenie	19
1 Wprowadzenie	23
Czym jest relacyjna baza danych?	23
Wstęp do SQL	24
Używanie SQL*Plus	25
Uruchamianie SQL*Plus	25
Uruchamianie SQL*Plus z wiersza poleceń	26
Wykonywanie instrukcji SELECT za pomocą SQL*Plus	26
SQL Developer	27
Tworzenie schematu bazy danych sklepu	30
Zawartość skryptu	30
Uruchamianie skryptu	31
Instrukcje DDL używane do tworzenia schematu bazy danych sklepu	32
Dodawanie, modyfikowanie i usuwanie wierszy	38
Dodawanie wiersza do tabeli	38
Modyfikowanie istniejącego wiersza w tabeli	39
Usuwanie wiersza z tabeli	40
Łączenie z bazą danych i rozłączanie	40
Kończenie pracy SQL*Plus	40
Wprowadzenie do Oracle PL/SQL	41
Podsumowanie	41
2 Pobieranie informacji z tabel bazy danych	43
Wykonywanie instrukcji SELECT dla jednej tabeli	43
Pobieranie wszystkich kolumn z tabeli	44
Wykorzystanie klauzuli WHERE do wskazywania wierszy do pobrania	44
Identyfikatory wierszy	44
Numery wierszy	45
Wykonywanie działań arytmetycznych	45
Wykonywanie obliczeń na danych	46
Korzystanie z kolumn w obliczeniach	47
Kolejność wykonywania działań	48
Używanie aliasów kolumn	48
Łączenie wartości z kolumn za pomocą konkatenacji	49
Wartości null	49
Wyświetlanie unikatowych wierszy	50
Porównywanie wartości	51
Operator <>	51
Operator >	52
Operator <=	52
Operator ANY	52
Operator ALL	52

6 Oracle Database 12c i SQL. Programowanie

Korzystanie z operatorów SQL	53
Operator LIKE	53
Operator IN	54
Operator BETWEEN	55
Operatory logiczne	55
Operator AND	55
Operator OR	56
Następstwo operatorów	56
Sortowanie wierszy za pomocą klauzuli ORDER BY	57
Instrukcje SELECT wykorzystujące dwie tabele	58
Używanie aliasów tabel	59
Iloczyny kartezjańskie	60
Instrukcje SELECT wykorzystujące więcej niż dwie tabele	60
Warunki złączenia i typy złączeń	61
Nierównozłączenia	61
Złączenia zewnętrzne	62
Złączenia własne	65
Wykonywanie złączeń za pomocą składni SQL/92	66
Wykonywanie złączeń wewnętrznych dwóch tabel z wykorzystaniem składni SQL/92	66
Upraszczanie złączeń za pomocą słowa kluczowego USING	67
Wykonywanie złączeń wewnętrznych obejmujących więcej niż dwie tabele (SQL/92)	67
Wykonywanie złączeń wewnętrznych z użyciem wielu kolumn (SQL/92)	68
Wykonywanie złączeń zewnętrznych z użyciem składni SQL/92	68
Wykonywanie złączeń własnych z użyciem składni SQL/92	69
Wykonywanie złączeń krzyżowych z użyciem składni SQL/92	70
Podsumowanie	70
3 SQL*Plus	71
Przeglądanie struktury tabeli	71
Edycja instrukcji SQL	72
Zapisywanie, odczytywanie i uruchamianie plików	73
Formatowanie kolumn	76
Ustawianie rozmiaru strony	77
Ustawianie rozmiaru wiersza	78
Czyszczenie formatowania kolumny	78
Użycie zmiennej	79
Zmienne tymczasowe	79
Zmienne zdefiniowane	81
Tworzenie prostych raportów	83
Użycie zmiennej tymczasowej w skrypcie	83
Użycie zmiennej zdefiniowanej w skrypcie	84
Przesyłanie wartości do zmiennej w skrypcie	84
Dodawanie nagłówka i stopki	85
Obliczanie sum pośrednich	86
Uzyskiwanie pomocy od SQL*Plus	87
Automatyczne generowanie instrukcji SQL	88
Kończenie połączenia z bazą danych i pracy SQL*Plus	88
Podsumowanie	89
4 Proste funkcje	91
Typy funkcji	91
Funkcje jednowierszowe	91
Funkcje znakowe	92
Funkcje numeryczne	98

Funkcje konwertujące	103
Funkcje wyrażeń regularnych	112
Funkcje agregujące	117
AVG()	118
COUNT()	119
MAX() i MIN()	119
STDDEV()	120
SUM()	120
VARIANCE()	120
Grupowanie wierszy	120
Grupowanie wierszy za pomocą klauzuli GROUP BY	120
Nieprawidłowe użycie funkcji agregujących	123
Filtrowanie grup wierszy za pomocą klauzuli HAVING	124
Jednoczesne używanie klauzul WHERE i GROUP BY	124
Jednoczesne używanie klauzul WHERE, GROUP BY i HAVING	125
Podsumowanie	125
5 Składowanie oraz przetwarzanie dat i czasu	127
Proste przykłady składowania i pobierania dat	127
Konwertowanie typów DataGodzina za pomocą funkcji TO_CHAR() i TO_DATE()	128
Konwersja daty i czasu na napis za pomocą funkcji TO_CHAR()	128
Konwersja napisu na wyrażenie DataGodzina za pomocą funkcji TO_DATE()	132
Ustawianie domyślnego formatu daty	134
Jak Oracle interpretuje lata dwucyfrowe?	135
Użycie formatu YY	135
Użycie formatu RR	136
Funkcje operujące na datach i godzinach	137
ADD_MONTHS()	138
LAST_DAY()	138
MONTHS_BETWEEN()	138
NEXT_DAY()	139
ROUND()	139
SYSDATE	140
TRUNC()	140
Strefy czasowe	140
Funkcje operujące na strefach czasowych	141
Strefa czasowa bazy danych i strefa czasowa sesji	141
Uzyskiwanie przesunięć strefy czasowej	142
Uzyskiwanie nazw stref czasowych	143
Konwertowanie wyrażenia DataGodzina z jednej strefy czasowej na inną	143
Datowniki (znaczniki czasu)	143
Typy datowników	144
Funkcje operujące na znacznikach czasu	147
Interwały czasowe	151
Typ INTERVAL YEAR TO MONTH	152
Typ INTERVAL DAY TO SECOND	153
Funkcje operujące na interwałach	155
Podsumowanie	156
6 Podzapytania	157
Rodzaje podzapytań	157
Pisanie podzapytań jednowierszowych	157
Podzapytania w klauzuli WHERE	157
Użycie innych operatorów jednowierszowych	158

8 Oracle Database 12c i SQL. Programowanie

Podzapytania w klauzuli HAVING	159
Podzapytania w klauzuli FROM (widoki wbudowane)	160
Błędy, które można napotkać	160
Pisanie podzapytań wielowierszowych	161
Użycie operatora IN z podzapytaniem wielowierszowym	161
Użycie operatora ANY z podzapytaniem wielowierszowym	162
Użycie operatora ALL z podzapytaniem wielowierszowym	163
Pisanie podzapytań wielokolumnowych	163
Pisanie podzapytań skorelowanych	163
Przykład podzapytania skorelowanego	163
Użycie operatorów EXISTS i NOT EXISTS z podzapytaniem skorelowanym	164
Pisanie zagnieżdżonych podzapytań	166
Pisanie instrukcji UPDATE i DELETE zawierających podzapytania	167
Pisanie instrukcji UPDATE zawierającej podzapytanie	167
Pisanie instrukcji DELETE zawierającej podzapytanie	168
Przygotowywanie podzapytań	168
Podsumowanie	169
7 Zapytania zaawansowane	171
Operatory zestawu	171
Przykładowe tabele	171
Operator UNION ALL	172
Operator UNION	173
Operator INTERSECT	174
Operator MINUS	174
Łączenie operatorów zestawu	175
Użycie funkcji TRANSLATE()	176
Użycie funkcji DECODE()	177
Użycie wyrażenia CASE	178
Proste wyrażenia CASE	179
Przeszukiwane wyrażenia CASE	179
Zapytania hierarchiczne	181
Przykładowe dane	181
Zastosowanie klauzul CONNECT BY i START WITH	182
Użycie pseudokolumny LEVEL	183
Formatowanie wyników zapytania hierarchicznego	183
Rozpoczynanie od węzła innego niż główny	184
Użycie podzapytania w klauzuli START WITH	185
Poruszanie się po drzewie w góre	185
Eliminowanie węzłów i gałęzi z zapytania hierarchicznego	185
Umieszczań innych warunków w zapytaniu hierarchicznym	186
Zapytania hierarchiczne wykorzystujące rekurencyjne podzapytania przygotowywane	187
Klauzule ROLLUP i CUBE	190
Przykładowe tabele	190
Użycie klauzuli ROLLUP	192
Klauzula CUBE	194
Funkcja GROUPING()	195
Klauzula GROUPING SETS	197
Użycie funkcji GROUPING_ID()	198
Kilkukrotne użycie kolumny w klauzuli GROUP BY	199
Użycie funkcji GROUP_ID()	200
Użycie CROSS APPLY i OUTER APPLY	201
CROSS APPLY	201
OUTER APPLY	202

LATERAL	202
Podsumowanie	203
8 Analiza danych	205
Funkcje analityczne	205
Przykładowa tabela	205
Użycie funkcji klasyfikujących	206
Użycie odwrotnych funkcji rankingowych	212
Użycie funkcji okna	212
Funkcje raportujące	218
Użycie funkcji LAG() i LEAD()	220
Użycie funkcji FIRST i LAST	221
Użycie funkcji regresji liniowej	221
Użycie funkcji hipotetycznego rankingu i rozkładu	222
Użycie klauzuli MODEL	223
Przykład zastosowania klauzuli MODEL	223
Dostęp do komórek za pomocą zapisu pozycyjnego i symbolicznego	224
Uzyskiwanie dostępu do zakresu komórek za pomocą BETWEEN i AND	225
Sieganie do wszystkich komórek za pomocą ANY i IS ANY	225
Pobieranie bieżącej wartości wymiaru za pomocą funkcji CURRENTV()	226
Uzyskiwanie dostępu do komórek za pomocą pętli FOR	227
Obsługa wartości NULL i brakujących	227
Modyfikowanie istniejących komórek	229
Użycie klauzuli PIVOT i UNPIVOT	230
Prosty przykład klauzuli PIVOT	230
Przedstawianie w oparciu o wiele kolumn	231
Użycie kilku funkcji agregujących w przedstawieniu	232
Użycie klauzuli UNPIVOT	233
Zapytania o określona liczbę wierszy	234
Użycie klauzuli FETCH FIRST	234
Użycie klauzuli OFFSET	235
Użycie klauzuli PERCENT	236
Użycie klauzuli WITH TIES	236
Odnajdywanie wzorców w danych	237
Odnajdywanie wzorców formacji typu V w danych z tabeli all_sales2	237
Odnajdywanie formacji typu W w danych z tabeli all_sales3	240
Odnajdywanie formacji typu V w tabeli all_sales3	241
Podsumowanie	242
9 Zmianianie zawartości tabeli	243
Wstawianie wierszy za pomocą instrukcji INSERT	243
Pomijanie listy kolumn	244
Określanie wartości NULL dla kolumny	244
Umieszczanie pojedynczych i podwójnych cudzysłów w wartościach kolumn	245
Kopiowanie wierszy z jednej tabeli do innej	245
Modyfikowanie wierszy za pomocą instrukcji UPDATE	245
Klauzula RETURNING	246
Usuwanie wierszy za pomocą instrukcji DELETE	246
Integralność bazy danych	247
Wymuszanie więzów klucza głównego	247
Wymuszanie więzów kluczy obcych	247
Użycie wartości domyślnych	248
Scalanie wierszy za pomocą instrukcji MERGE	249

10 Oracle Database 12c i SQL. Programowanie

Transakcje bazodanowe	251
Zatwierdzanie i wycofywanie transakcji	251
Rozpoczynanie i kończenie transakcji	252
Punkty zachowania	252
ACID — właściwości transakcji	254
Transakcje współbieżne	254
Blokowanie transakcji	255
Poziomy izolacji transakcji	256
Przykład transakcji SERIALIZABLE	256
Zapytania retrospektywne	257
Przyznawanie uprawnień do używania zapytań retrospektywnych	257
Zapytania retrospektywne w oparciu o czas	258
Zapytania retrospektywne z użyciem SCN	259
Podsumowanie	260
10 Użytkownicy, uprawnienia i role	261
Bardzo krótkie wprowadzenie do przechowywania danych	261
Użytkownicy	262
Tworzenie konta użytkownika	262
Zmienianie hasła użytkownika	263
Usuwanie konta użytkownika	263
Uprawnienia systemowe	263
Przyznawanie uprawnień systemowych użytkownikowi	263
Sprawdzanie uprawnień systemowych przyznanych użytkownikowi	264
Zastosowanie uprawnień systemowych	265
Odbieranie uprawnień systemowych	265
Uprawnienia obiektowe	266
Przyznawanie użytkownikowi uprawnień obiektowych	266
Sprawdzanie przekazanych uprawnień	267
Sprawdzanie otrzymanych uprawnień obiektowych	268
Zastosowanie uprawnień obiektowych	269
Synonimy	270
Synonimy publiczne	270
Odbieranie uprawnień obiektowych	271
Role	271
Tworzenie ról	271
Przyznawanie uprawnień roli	272
Przyznawanie roli użytkownikowi	272
Sprawdzanie ról przyznanych użytkownikowi	272
Sprawdzanie uprawnień systemowych przyznanych roli	273
Sprawdzanie uprawnień obiektowych przyznanych roli	274
Zastosowanie uprawnień przyznanych roli	275
Aktywacja i deaktywacja ról	276
Odbieranie roli	276
Odbieranie uprawnień roli	276
Usuwanie roli	277
Obserwacja	277
Uprawnienia wymagane do przeprowadzania obserwacji	277
Przykłady obserwacji	277
Perspektywy zapisu obserwacji	279
Podsumowanie	279

11 Tworzenie tabel, sekwencji, indeksów i perspektyw	281
Tabele	281
Tworzenie tabeli	281
Pobieranie informacji o tabelach	282
Uzyskiwanie informacji o kolumnach w tabeli	283
Zmienianie tabeli	284
Zmienianie nazwy tabeli	291
Dodawanie komentarza do tabeli	291
Obcinanie tabeli	292
Usuwanie tabeli	292
Typy BINARY_FLOAT i BINARY_DOUBLE	292
Użycie kolumn DEFAULT ON NULL	293
Kolumny niewidoczne	294
Sekwencje	296
Tworzenie sekwencji	296
Pobieranie informacji o sekwencjach	298
Używanie sekwencji	298
Wypełnianie klucza głównego z użyciem sekwencji	300
Określanie domyślnej wartości kolumny za pomocą sekwencji	300
Kolumny typu IDENTITY	301
Modyfikowanie sekwencji	301
Usuwanie sekwencji	302
Indeksy	302
Tworzenie indeksu typu B-drzewo	303
Tworzenie indeksów opartych na funkcjach	303
Pobieranie informacji o indeksach	304
Pobieranie informacji o indeksach kolumny	304
Modyfikowanie indeksu	305
Usuwanie indeksu	305
Tworzenie indeksu bitmapowego	305
Perspektywy	306
Tworzenie i używanie perspektyw	307
Modyfikowanie perspektywy	313
Usuwanie perspektywy	313
Używanie niewidocznych kolumn w perspektywach	313
Archiwa migawek	314
Podsumowanie	316
12 Wprowadzenie do programowania w PL/SQL	317
Blok	317
Zmienne i typy	319
Logika warunkowa	319
Pętle	320
Proste pętle	320
Pętle WHILE	321
Pętle FOR	321
Kursory	322
Krok 1. — deklarowanie zmiennych przechowujących wartości kolumn	322
Krok 2. — deklaracja kursora	322
Krok 3. — otwarcie kursora	323
Krok 4. — pobieranie wierszy z kursora	323
Krok 5. — zamknięcie kursora	323
Pełny przykład — product_cursor.sql	324

12 Oracle Database 12c i SQL. Programowanie

Kursory i pętle FOR	325
Instrukcja OPEN-FOR	325
Kursory bez ograniczenia	327
Wyjątki	328
Wyjątek ZERO_DIVIDE	330
Wyjątek DUP_VAL_ON_INDEX	330
Wyjątek INVALID_NUMBER	330
Wyjątek OTHERS	331
Procedury	331
Tworzenie procedury	332
Wywoływanie procedury	333
Uzyskiwanie informacji o procedurach	334
Usuwanie procedury	335
Przeglądanie błędów w procedurze	335
Funkcje	335
Tworzenie funkcji	336
Wywoływanie funkcji	336
Uzyskiwanie informacji o funkcjach	337
Usuwanie funkcji	337
Pakiety	337
Tworzenie specyfikacji pakietu	338
Tworzenie treści pakietu	338
Wywoływanie funkcji i procedur z pakietu	339
Uzyskiwanie informacji o funkcjach i procedurach w pakiecie	340
Usuwanie pakietu	340
Wyzwalacze	340
Kiedy uruchamiany jest wyzwalacz	340
Przygotowania do przykładu wyzwalacza	341
Tworzenie wyzwalacza	341
Uruchamianie wyzwalacza	343
Uzyskiwanie informacji o wyzwalaczach	343
Włączanie i wyłączanie wyzwalacza	345
Usuwanie wyzwalacza	345
Rozszerzenia PL/SQL	345
Typ SIMPLE_INTEGER	345
Sekwencje w PL/SQL	346
Generowanie natywnego kodu maszynowego z PL/SQL	347
Klauzula WITH	347
Podsumowanie	348
13 Obiekty bazy danych	349
Wprowadzenie do obiektów	349
Uruchomienie skryptu tworzącego schemat bazy danych object_schema	350
Tworzenie typów obiektowych	350
Uzyskiwanie informacji o typach obiektowych za pomocą DESCRIBE	351
Użycie typów obiektowych w tabelach bazy danych	352
Obiekty kolumnowe	352
Tabele obiektowe	354
Identyfikatory obiektów i odwołania obiektowe	357
Porównywanie wartości obiektów	359
Użycie obiektów w PL/SQL	361
Funkcja get_products()	361
Procedura display_product()	362
Procedura insert_product()	363

Procedura update_product_price()	363
Funkcja get_product()	364
Procedura update_product()	364
Funkcja get_product_ref()	365
Procedura delete_product()	365
Procedura product_lifecycle()	366
Procedura product_lifecycle2()	367
Dziedziczenie typów	368
Uruchamianie skryptu tworzącego schemat bazy danych object_schema2	368
Dziedziczenie atrybutów	369
Użycie podtypu zamiast typu nadrzednego	370
Przykłady SQL	370
Przykłady PL/SQL	371
Obiekty NOT SUBSTITUTABLE	371
Inne przydatne funkcje obiektów	372
Funkcja IS OF()	372
Funkcja TREAT()	375
Funkcja SYS_TYPEID()	378
Typy obiektowe NOT INSTANTIABLE	378
Konstruktory definiowane przez użytkownika	379
Przesłanianie metod	382
Uogólnione wywoływanie	384
Uruchomienie skryptu tworzącego schemat bazy danych object_schema3	384
Dziedziczenie atrybutów	384
Podsumowanie	385
14 Kolekcje	387
Podstawowe informacje o kolekcjach	387
Uruchomienie skryptu tworzącego schemat bazy danych collection_schema	387
Tworzenie kolekcji	388
Tworzenie typu VARRAY	388
Tworzenie tabeli zagnieżdzonej	388
Użycie kolekcji do definiowania kolumn w tabeli	389
Użycie typu VARRAY do zdefiniowania kolumny w tabeli	389
Użycie typu tabeli zagnieżdzonej do zdefiniowania kolumny w tabeli	389
Uzyskiwanie informacji o kolekcjach	389
Uzyskiwanie informacji o tablicy VARRAY	389
Uzyskiwanie informacji o tabeli zagnieżdzonej	390
Umieszczanie elementów w kolekcji	392
Umieszczanie elementów w tablicy VARRAY	392
Umieszczanie elementów w tabeli zagnieżdzonej	392
Pobieranie elementów z kolekcji	392
Pobieranie elementów z tablicy VARRAY	393
Pobieranie elementów z tabeli zagnieżdzonej	393
Użycie funkcji TABLE() do interpretacji kolekcji jako serii wierszy	394
Użycie funkcji TABLE() z typem VARRAY	394
Użycie funkcji TABLE() z tabelą zagnieżdzoną	395
Modyfikowanie elementów kolekcji	395
Modyfikowanie elementów tablicy VARRAY	396
Modyfikowanie elementów tabeli zagnieżdzonej	396
Użycie metody mapującej do porównywania zawartości tabel zagnieżdżonych	397
Użycie funkcji CAST do konwersji kolekcji z jednego typu na inny	399
Użycie funkcji CAST() do konwersji tablicy VARRAY na tabelę zagnieżdzoną	399
Użycie funkcji CAST() do konwersji tabeli zagnieżdzonej na tablicę VARRAY	400

Użycie kolekcji w PL/SQL	400
Manipulowanie tablicą VARRAY	400
Manipulowanie tabelą zagnieżdzoną	402
Metody operujące na kolekcjach w PL/SQL	403
Kolekcje wielopoziomowe	411
Uruchomienie skryptu tworzącego schemat bazy danych collection_schema2	412
Korzystanie z kolekcji wielopoziomowych	412
Rozszerzenia kolekcji wprowadzone w Oracle Database 10g	414
Uruchomienie skryptu tworzącego schemat bazy danych collection_schema3	414
Tablice asocjacyjne	415
Zmienianie rozmiaru typu elementu	415
Zwiększanie liczby elementów w tablicy VARRAY	416
Użycie tablic VARRAY w tabelach tymczasowych	416
Użycie innej przestrzeni tabel dla tabeli składającej tabelę zagnieżdzoną	416
Obsługa tabel zagnieżdzonych w standardzie ANSI	417
Podsumowanie	424
15 Duże obiekty	425
Podstawowe informacje o dużych obiektach (LOB)	425
Przykładowe pliki	425
Rodzaje dużych obiektów	426
Tworzenie tabel zawierających duże obiekty	427
Użycie dużych obiektów w SQL	428
Użycie obiektów CLOB i BLOB	428
Użycie obiektów BFILE	430
Użycie dużych obiektów w PL/SQL	431
APPEND()	433
CLOSE()	433
COMPARE()	434
COPY()	435
CREATETEMPORARY()	435
ERASE()	436
FILECLOSE()	436
FILECLOSEALL()	437
FILEEXISTS()	437
FILEGETNAME()	437
FILEISOPEN()	438
FILEOPEN()	438
FREETEMPORARY()	439
GETCHUNKSIZE()	439
GETLENGTH()	439
GET_STORAGE_LIMIT()	440
INSTR()	440
ISOPEN()	441
ISTEMPORARY()	441
LOADFROMFILE()	442
LOADBLOBFROMFILE()	443
LOADCLOBFROMFILE()	443
OPEN()	444
READ()	445
SUBSTR()	445
TRIM()	446

WRITE()	447
WRITEAPPEND()	447
Przykładowe procedury PL/SQL	448
Typy LONG i LONG RAW	462
Przykładowe tabele	462
Wstawianie danych do kolumn typu LONG i LONG RAW	462
Przekształcanie kolumn LONG i LONG RAW w duże obiekty	463
Nowe właściwości dużych obiektów w Oracle Database 10g	463
Niejawnna konwersja między obiektami CLOB i NCLOB	464
Użycie atrybutu :new, gdy obiekt LOB jest używany w wyzwalaczu	464
Nowe właściwości dużych obiektów w Oracle Database 11g	465
Szyfrowanie danych LOB	465
Kompresja danych LOB	469
Usuwanie powtarzających się danych LOB	469
Nowe właściwości dużych obiektów w Oracle Database 12c	469
Podsumowanie	470
16 Optymalizacja SQL	471
Podstawowe informacje o optymalizacji SQL	471
Należy filtrować wiersze za pomocą klauzuli WHERE	471
Należy używać złączeń tabel zamiast wielu zapytań	472
Wykonując złączenia, należy używać w pełni kwalifikowanych odwołań do kolumn	473
Należy używać wyrażeń CASE zamiast wielu zapytań	473
Należy dodać indeksy do tabel	474
Kiedy tworzyć indeks typu B-drzewo	475
Kiedy tworzyć indeks bitmapowy	475
Należy stosować klauzulę WHERE zamiast HAVING	475
Należy używać UNION ALL zamiast UNION	476
Należy używać EXISTS zamiast IN	477
Należy używać EXISTS zamiast DISTINCT	477
Należy używać GROUPING SETS zamiast CUBE	478
Należy stosować zmienne dowiązane	478
Nieidentyczne instrukcje SQL	478
Identyczne instrukcje SQL korzystające ze zmiennych dowiązanych	478
Wypisywanie listy i wartości zmiennych dowiązanych	479
Użycie zmiennej dowiązanej do składowania wartości zwróconej przez funkcję PL/SQL	480
Użycie zmiennej dowiązanej do składowania wierszy z REFCURSOR	480
Porównywanie kosztu wykonania zapytań	480
Przeglądanie planów wykonania	481
Porównywanie planów wykonania	485
Przesyłanie wskazówek do optymalizatora	486
Dodatkowe narzędzia optymalizujące	487
Oracle Enterprise Manager	487
Automatic Database Diagnostic Monitor	488
SQL Tuning Advisor	488
SQL Access Advisor	488
SQL Performance Analyzer	488
Database Replay	488
Real-Time SQL Monitoring	488
SQL Plan Management	489
Podsumowanie	489

16 Oracle Database 12c i SQL. Programowanie

17 XML i baza danych Oracle	491
Wprowadzenie do XML	491
Generowanie XML z danych relacyjnych	492
XMLEMENT()	492
XMLATTRIBUTES()	494
XMLFOREST()	494
XMLAGG()	495
XMLCOLATTVAL()	497
XMLCONCAT()	498
XMLPARSE()	498
XMLPI()	499
XMLCOMMENT()	499
XMLSEQUENCE()	500
XMLSERIALIZE()	501
Przykład zapisywania danych XML do pliku w PL/SQL	501
XMLQUERY()	502
Zapisywanie XML w bazie danych	506
Przykładowy plik XML	506
Tworzenie przykładowego schematu XML	506
Pobieranie informacji z przykładowego schematu XML	508
Aktualizowanie informacji w przykładowym schemacie XML	511
Podsumowanie	514
A Typy danych Oracle	515
Typy w Oracle SQL	515
Typy w Oracle PL/SQL	517
Skorowidz	519

O autorze

Jason Price jest niezależnym konsultantem i byłym kierownikiem projektu w Oracle Corporation. Brał udział w pracach nad wieloma produktami Oracle, w tym bazą danych, serwerem aplikacji i kilkoma aplikacjami CRM. Jest certyfikowanym administratorem i programistą baz danych Oracle i posiada ponad 15-letnie doświadczenie w branży oprogramowania. Napisał wiele książek na temat Oracle, Javy i .NET. Uzyskał tytuł licencjata (z wyróżnieniem) w dziedzinie fizyki na brytyjskim University of Bristol.

Podziękowania

Dziękuję wspaniałym ludziom z McGraw-Hill Education/Professional. Dziękuję też Scottowi Mikolaitisowi i Nidhi Choprze.

Wprowadzenie

Współcześnie dostęp do systemów zarządzania bazami danych jest realizowany z użyciem standardowego języka Structured Query Language (strukturalnego języka zapytań), czyli SQL. SQL umożliwia między innymi pobieranie, wstawianie, modyfikowanie i usuwanie informacji z bazy danych. Ta książka pozwala dobrze opanować język SQL, a ponadto zawiera wiele praktycznych przykładów. Wszystkie skrypty i programy prezentowane w książce są dostępne online (więcej informacji na ten temat znajduje się w podroziale „Pobieranie przykładów”).

Dzięki tej książce:

- Opanujesz standardowy SQL, a także jego rozszerzenia opracowane przez Oracle Corporation, umożliwiające wykorzystanie specyficznych właściwości bazy danych Oracle.
- Poznasz język PL/SQL (Procedural Language/SQL), który wywodzi się z SQL i umożliwia pisanie programów zawierających instrukcje SQL.
- Dowiesz się, jak używać SQL*Plus do uruchamiania instrukcji SQL, skryptów i raportów. SQL*Plus jest narzędziem umożliwiającym interakcję z bazą danych.
- Dowiesz się, jak wykonywać zapytania, wstawiać, modyfikować i usuwać dane z bazy danych.
- Opanujesz tworzenie tabel, sekwencji, indeksów, perspektyw i kont użytkowników.
- Dowiesz się, jak wykonywać transakcje zawierające wiele instrukcji SQL.
- Opanujesz definiowanie typów obiektowych i tworzenie tabel obiektowych do obsługi danych zaawansowanych.
- Nauczysz się wykorzystywać duże obiekty do obsługi plików multimedialnych zawierających obrazy, muzykę i filmy.
- Dowiesz się, jak wykonywać skomplikowane obliczenia za pomocą funkcji analitycznych.
- Opanujesz wysoko wydajne techniki optymalizacyjne, znaczco przyspieszające wykonywanie instrukcji SQL.
- Poznasz możliwości obsługi XML w bazie danych Oracle.
- Dowiesz się, jak wykorzystywać najnowsze możliwości języka SQL wprowadzone w Oracle Database 12c.

Książka zawiera 17 rozdziałów i dodatek.

Rozdział 1. „Wprowadzenie”

W tym rozdziale znajduje się opis relacyjnych baz danych, wprowadzenie do SQL i kilka przykładowych zapytań. Nauczysz się w nim również używać SQL*Plus i SQL Developer do wykonywania zapytań, a także krótko omówimy PL/SQL.

Rozdział 2. „Pobieranie informacji z tabel bazy danych”

Dowiesz się, jak pobrać informacje z jednej lub kilku tabel, korzystając z instrukcji SELECT. Nauczysz się również używać wyrażeń arytmetycznych do wykonywania obliczeń. Poznasz klauzulę WHERE, umożliwiającą filtrowanie wierszy, a także dowiesz się, jak je sortować.

Rozdział 3. „SQL*Plus”

W tym rozdziale użyjemy SQL*Plus do przejrzenia struktury tabeli, edytowania instrukcji SQL, zapisywania i uruchamiania skryptów, formatowania kolumn wyników. Nauczysz się również używać zmiennych i generować raporty.

Rozdział 4. „Proste funkcje”

W tym rozdziale poznasz kilka funkcji wbudowanych do bazy danych Oracle. Funkcja może przyjmować parametry wejściowe i zwraca parametr wyjściowy. Funkcje umożliwiają między innymi obliczanie średnich i pierwiastków kwadratowych.

Rozdział 5. „Składowanie oraz przetwarzanie dat i czasu”

Dowiesz się, w jaki sposób baza danych Oracle przetwarza oraz składuje daty i czas. Poznasz również datowniki umożliwiające składowanie określonej daty i czasu, a także interwały czasowe umożliwiające składowanie okresu.

Rozdział 6. „Podzapytania”

Dowiesz się, w jaki sposób można umieścić instrukcję SELECT w zewnętrznej instrukcji SQL. Wewnętrzna instrukcję SQL nazywamy podzapytaniem. Poznasz różne rodzaje podzapytań i zobaczy, jak umożliwiają one tworzenie złożonych instrukcji z prostych składników.

Rozdział 7. „Zapytania zaawansowane”

Dowiesz się, jak wykonywać zapytania zawierające zaawansowane operatory i funkcje, takie jak: operatory zestawu łączące wiersze zwracane przez kilka zapytań, funkcja TRANSLATE() konwertująca znaki w jednym napisie na znaki w innym napisie, funkcja DECODE() wyszukująca wartość w zestawie wartości, wyrażenie CASE wykonujące logikę if-then-else oraz klauzule ROLLUP i CUBE zwracające wiersze zawierające sumy częstek. Nowe w Oracle Database 12c klauzule CROSS APPLY i OUTER APPLY łączą wiersze z dwóch wyrażeń SELECT, a LATERAL zwraca wbudowany widok danych.

Rozdział 8. „Analiza danych”

Poznasz funkcje analityczne umożliwiające wykonywanie złożonych obliczeń, takich jak wyszukiwanie najlepiej sprzedającego się produktu w poszczególnych miesiącach, najlepszych sprzedawców itd. Dowiesz się, jak wykonywać zapytania o dane uporządkowane hierarchicznie. Poznasz również klauzulę MODEL wykonującą obliczenia międzywierszowe oraz klauzule PIVOT i UNPIVOT, które umożliwiają poznanie ogólnych trendów w dużych ilościach danych. Nowe w Oracle Database 12c klauzule to MATCH_RECOGNIZE umożliwiająca odnalezienie wzorca w danych i FETCH_FIRST umożliwiająca wykonanie zapytań zwracających N pierwszych wierszy wyniku.

Rozdział 9. „Zmienianie zawartości tabeli”

Dowiesz się, jak wstawiać, modyfikować i usuwać wiersze za pomocą instrukcji INSERT, UPDATE i DELETE oraz jak utrzymać wynik transakcji, korzystając z instrukcji COMMIT lub wycofać wyniki transakcji za pomocą instrukcji ROLLBACK. Dowiesz się również, w jaki sposób baza danych Oracle obsługuje kilka transakcji wykonywanych w tym samym momencie.

Rozdział 10. „Użytkownicy, uprawnienia i role”

Dowiesz się, czym są użytkownicy bazy danych oraz jak uprawnienia i role umożliwiają określenie czynności, które może wykonać użytkownik w bazie danych.

Rozdział 11. „Tworzenie tabel, sekwencji, indeksów i perspektyw”

Poznasz tabele i sekwencje, które generują serie liczb, a także indeksy, które przypominają indeksy w książkach i umożliwiają szybkie uzyskanie dostępu do wierszy. Dowiesz się również czegoś o perspektywach, które są wstępnie zdefiniowanymi zapytaniami jednej lub kilku tabel. Do zalet perspektyw możemy zaliczyć

czyć to, że umożliwiają one ukrycie złożoności przed użytkownikiem, a także implementują kolejny poziom zabezpieczeń, zezwalając na przeglądanie jedynie ograniczonego zestawu danych z tabeli. Poznasz również archiwum migawek. W archiwum migawek są składowane zmiany dokonane w tabeli w pewnym okresie. Nowością w Oracle Database 12c jest możliwość definiowania widocznych i niewidocznych kolumn w tabeli.

Rozdział 12. „Wprowadzenie do programowania w PL/SQL”

W tym rozdziale poznasz język PL/SQL, zbudowany na podstawie SQL i umożliwiający pisanie programów składowanych w bazie danych oraz zawierających instrukcje SQL. Język ten posiada standardowe konstrukty programistyczne.

Rozdział 13. „Obiekty bazy danych”

Dowiesz się, jak tworzyć typy obiektowe w bazie danych, które mogą zawierać atrybuty i metody. Za pomocą tych typów obiektowych zdefiniujemy obiekty kolumnowe i tabele obiektów, a także nauczysz się manipulować obiektami za pomocą SQL i PL/SQL.

Rozdział 14. „Kolekcje”

Dowiesz się, jak tworzyć typy kolekcji, które mogą zawierać wiele elementów. Użyjemy kolekcji do definiowania kolumn w tabelach. Nauczysz się również manipulować kolekcjami za pomocą SQL i PL/SQL.

Rozdział 15. „Duże obiekty”

Poznasz duże obiekty, które mogą przechowywać do 128 terabajtów danych znakowych i binarnych lub wskazywać na plik zewnętrzny, oraz starsze typy LONG, obsługiwane przez Oracle Database 12c w celu zachowania kompatybilności z wcześniejszymi wersjami.

Rozdział 16. „Optymalizacja SQL”

Ten rozdział zawiera kilka wskazówek pozwalających skrócić czas wykonywania zapytań. Dowiesz się również czegoś o optymalizatorze Oracle, a także możliwości przesyłania wskazówek do optymalizatora. Wprowadzone zostaną też zaawansowane narzędzia do optymalizacji.

Rozdział 17. „XML i baza danych Oracle”

Extensible Markup Language (XML) jest językiem znaczników ogólnego przeznaczenia. XML umożliwia przesyłanie ustrukturyzowanych danych w internecie i może być używany do kodowania danych i innych dokumentów. Z tego rozdziału dowiesz się, jak generować kod XML na podstawie danych relacyjnych oraz jak zapisać kod XML w bazie danych.

Dodatek A

Dodatek zawiera opis typów danych dostępnych w Oracle SQL i PL/SQL.

Docelowa grupa czytelników

Ta książka jest odpowiednia dla następujących czytelników:

- programistów, którzy chcą pisać w SQL i PL/SQL,
- administratorów baz danych wymagających dogłębnej znajomości SQL,
- użytkowników biznesowych, którzy muszą pisać zapytania SQL w celu pobrania informacji z bazy danych organizacji,
- kierowników technicznych lub konsultantów, którzy potrzebują wprowadzenia do SQL i PL/SQL.

Nie jest konieczna wcześniejsza znajomość SQL i PL/SQL. Wszystko, co jest potrzebne do biegłego opanowania SQL i PL/SQL, można znaleźć w tej książce.

Pobieranie przykładów

Wszystkie skrypty, programy i inne pliki używane w tej książce można pobrać z serwera FTP Wydawnictwa Helion, pod adresem: <ftp://ftp.helion.pl/przykłady/ord12p.zip>. Pliki są umieszczone w archiwum ZIP — po jego rozpakowaniu tworzony jest katalog `sql_book`, w którym znajdują się następujące podkatalogi:

- *pliki* zawierający przykładowe pliki używane w rozdziale 14.,
- *SQL* zawierający skrypty SQL używane w całej książce, w tym skrypty tworzące i umieszczające dane w przykładowych tabelach,
- *XML* zawierający pliki XML używane w rozdziale 17.

Mam nadzieję, że spodoba Ci się ta książka!

ROZDZIAŁ

1

Wprowadzenie

W tym rozdziale poznamy informacje na temat:

- relacyjnych baz danych,
- strukturalnego języka zapytań (SQL — *Structured Query Language*), używanego w pracy z bazami danych,
- SQL*Plus — interaktywnego narzędzia tekstowego do uruchamiania instrukcji SQL, utworzonego przez Oracle,
- SQL Developer — graficznego narzędzia do tworzenia baz danych,
- PL/SQL — utworzonego przez Oracle proceduralnego języka programowania, który umożliwia tworzenie programów.

Czym jest relacyjna baza danych?

Założenia relacyjnych baz danych zostały opracowane w 1970 roku przez dr. E.F. Codda. Opisał on teorię relacyjnych baz danych w artykule *A Relational Model of Data for Large Shared Data Banks*, opublikowanym w „Communications of the Association for Computing Machinery” (t. 13, nr 6, czerwiec 1970).

Podstawowe założenia relacyjnego modelu baz danych są stosunkowo łatwe do zrozumienia. **Relacyjna baza danych** jest zbiorem powiązanych informacji, umieszczonych w tabelach. Dane w tabeli są przechowywane w **wierszach** i uporządkowane w **kolumnach**. Tabele są przechowywane w **schematach** baz danych, czyli obszarach, w których użytkownicy mogą przechowywać swoje tabele. Użytkownik może przypisywać **uprawnienia**, dzięki którym inne osoby będą mogły uzyskać do nich dostęp.

Dane często porządkuje się w tabelach — ceny akcji czy rozkłady jazdy pociągów. W jednej z przykładowych tabel w tej książce będą zapisywane informacje o klientach fikcyjnego sklepu: ich imiona, nazwiska, daty urodzenia (dob — ang. *date of birth*) i numery telefonów.

first_name	last_name	dob	phone
Jan	Nikiel	65/01/01	800-555-1211
Lidia	Stal	68/02/05	800-555-1212
Stefan	Brąz	71/03/16	800-555-1213
Grażyna	Cynk		800-555-1214
Jadwiga	Mosiądz	70/05/20	

Taka tabela może być przechowywana:

- jako tabela w bazie danych,
- jako plik HTML na stronie internetowej,
- na karcie w pudełku.

Należy pamiętać, że informacje tworzące bazę danych są czymś innym niż system wykorzystywany do uzyskiwania do nich dostępu. Oprogramowanie używane do pracy z bazą danych nazywamy **systemem zarządzania bazą danych**. Baza danych Oracle jest właśnie takim systemem, a inne programy tego typu to Microsoft SQL Server, DB2 i MySQL.

W każdej bazie danych musi istnieć jakiś sposób wprowadzania i pobierania informacji, najlepiej korzystający z popularnego języka zrozumiałego dla wszystkich baz danych. Systemy zarządzania bazą danych implementują taki standardowy język nazywany **strukturalnym językiem zapytań**, czyli SQL (*Structured Query Language*). Umożliwia on pobieranie, dodawanie, modyfikowanie i usuwanie informacji z bazy danych.

Wstęp do SQL

Strukturalny język zapytań (SQL — *Structured Query Language*) jest standardowym językiem zaprojektowanym do pracy z relacyjnymi bazami danych.



Uwaga Według American National Standards Institute, „es kju el” jest prawidłowym sposobem odczytywania skrótu SQL. Często jednak słyszy się również angielskie słowo *sequel* („sikleł”).

Pierwsza implementacja SQL opartego na przełomowej pracy dr. E.F. Codda została opracowana przez IBM w połowie lat 70. Firma prowadziła projekt badawczy o nazwie System R i podczas jego realizacji opracowano SQL. W 1979 roku firma Relational Software Inc. (dziś znana jako Oracle Corporation) opublikowała pierwszą komercyjną wersję SQL.

SQL został uznany za standard przez American National Standards Institute (ANSI) w 1986 roku, ale implementacje różnych firm różnią się od siebie.

W SQL jest wykorzystywana prosta składnia, której z łatwością można się nauczyć. Proste przykłady zastosowania jej zostaną zaprezentowane w tym rozdziale. Wyróżniamy pięć typów instrukcji SQL:

- **Zapytania** pobierają wiersze przechowywane w tabelach bazy danych. Do utworzenia zapytania wykorzystujemy instrukcję SELECT.
- **Instrukcje DML (Data Manipulation Language)** służą do modyfikowania zawartości tabel. Istnieją trzy takie instrukcje:
 - **INSERT** dodaje wiersze do tabeli.
 - **UPDATE** zmienia wiersze.
 - **DELETE** usuwa wiersze.
- **Instrukcje DDL (Data Definition Language)** definiują struktury danych, takie jak tabele tworzące bazę danych. Wyróżniamy pięć podstawowych typów instrukcji DDL:
 - **CREATE** tworzy strukturę bazy danych, na przykład instrukcja CREATE TABLE służy do tworzenia tabeli, a CREATE USER jest wykorzystywana do tworzenia użytkownika bazy danych.
 - **ALTER** modyfikuje strukturę bazy danych, na przykład ALTER TABLE służy do modyfikacji tabeli.
 - **DROP** usuwa strukturę bazy danych, na przykład DROP TABLE służy do usuwania tabeli.
 - **RENAME** zmienia nazwę tabeli.
 - **TRUNCATE** usuwa wszystkie wiersze z tabeli.
- **Instrukcje TC (Transaction Control)** albo trwale zapisują zmiany wprowadzone w wierszach, albo je cofają. Wyróżniamy trzy instrukcje TC:
 - **COMMIT** trwale zapisuje zmiany wprowadzone do wierszy.
 - **ROLLBACK** cofa zmiany dokonane w wierszach.
 - **SAVEPOINT** tworzy punkt zachowania, do którego można cofnąć zmiany.
- **Instrukcje DCL (Data Control Language)** służą do nadawania uprawnień dostępu do struktur bazy danych. Istnieją dwie instrukcje DCL:

- **GRANT** daje użytkownikowi dostęp do wskazanej struktury bazy danych.
- **REVOKE** odbiera użytkownikowi prawo dostępu do wskazanej struktury bazy danych.

Oracle dostarcza program SQL*Plus umożliwiający wprowadzenie instrukcji SQL i uzyskanie rezultatów ich działania z bazy danych. SQL*Plus umożliwia również wykonanie skryptów zawierających instrukcje SQL oraz poleceń SQL*Plus.

Jest wiele sposobów uruchamiania instrukcji SQL i pobierania wyników z bazy danych. Można to zrobić za pomocą oprogramowania Oracle Forms and Reports lub programów napisanych w innych językach, takich jak Java i C#.Więcej informacji na temat wykonywania instrukcji SQL w programach pisanych w języku Java można znaleźć w mojej książce *Oracle9i JDBC Programming* (Oracle Press, 2002). Więcej informacji na temat wykonywania instrukcji SQL w programach pisanych w języku C# można znaleźć w mojej książce *Mastering C# Database Programming* (Sybex, 2003).

Używanie SQL*Plus

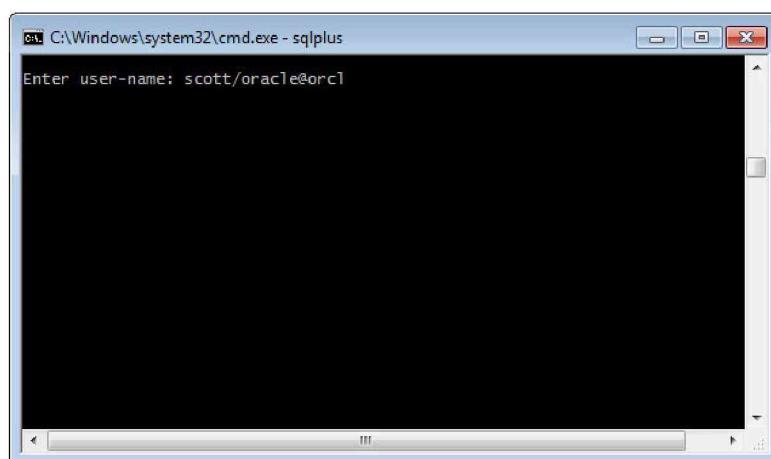
Z kolejnego podrozdziału dowiesz się, jak uruchomić program SQL*Plus i przesyłać zapytanie do bazy danych.

Uruchamianie SQL*Plus

Jeżeli używasz Windows 7, możesz uruchomić SQL*Plus, wybierając *Wszystkie programy/Oracle/Application Development/SQL Plus*. Jeżeli używasz systemu Unix lub Linux, SQL*Plus uruchomisz, wpisując `sqlplus` z wiersza poleceń.

Rysunek 1.1 prezentuje SQL*Plus uruchomiony w systemie Windows 7.

Rysunek 1.1.
Oracle Database 12c
SQL*Plus uruchomiony
w Windows 7



Na rysunku przedstawiono proces łączenia się użytkownika `scott` z bazą danych. Użytkownik `scott` jest tworzony w wielu bazach danych Oracle, a jego hasło w mojej bazie danych to `oracle`.

Nazwa hosta, znajdująca się po znaku @, informuje program SQL*Plus o tym, gdzie została uruchomiona baza danych. Jeżeli oprogramowanie działa na komputerze lokalnym, zwykle nazwa hosta jest pomijana (czyli wpisujemy `scott/oracle`) — w takim przypadku SQL*Plus próbuje połączyć się z bazą danych na tym samym komputerze, na którym jest uruchomiony. Jeżeli baza danych nie jest uruchomiona na komputerze lokalnym, należy uzyskać nazwę hosta od jej administratora (DBA).

Jeśli konto `scott` nie istnieje lub jest zablokowane, należy poprosić administratora o inną nazwę użytkownika i hasło. W przykładach z pierwszej części rozdziału można korzystać z dowolnego konta użytkownika.

Uruchamianie SQL*Plus z wiersza poleceń

Program SQL*Plus można uruchomić również z wiersza poleceń, za pomocą wyrażenia `sqlplus`. Pełna składnia tego polecenia ma postać:

```
sqlplus [nazwa_użytkownika[/hasło[@nazwa_hosta]]]
```

gdzie:

- *nazwa_użytkownika* oznacza nazwę użytkownika bazy danych,
- *hasło* oznacza hasło użytkownika bazy danych,
- *nazwa_hosta* oznacza bazę danych, z którą chcemy się połączyć.

Poniżej przedstawiono przykłady użycia polecenia `sqlplus`:

```
sqlplus scott/oracle
sqlplus scott/oracle@orcl
```

Jeżeli SQL*Plus używasz w systemie Windows, instalator Oracle doda automatycznie katalog programu SQL*Plus do ścieżki systemowej. Jeśli używasz systemu Unix lub Linux, masz dwa sposoby, by uruchomić SQL*Plus:

- za pomocą polecenia `cd` przejdź do katalogu, w którym znajduje się plik wykonywalny `sqlplus`, i uruchom `sqlplus` z tego katalogu;
- dodaj katalog, w którym znajduje się `sqlplus`, do systemowej ścieżki dostępu i uruchom `sqlplus`. W razie problemów z ustawianiem systemowej ścieżki dostępu należy się skontaktować z administratorem systemu.

Ze względów bezpieczeństwa można ukryć hasło podczas łączenia się z bazą danych. Można na przykład wpisać:

```
sqlplus scott@orcl
```

W takiej sytuacji SQL*Plus poprosi o wprowadzenie hasła, które pozostanie niewidoczne.

Można również po prostu wpisać:

```
sqlplus
```

W takim przypadku SQL*Plus poprosi o wprowadzenie nazwy użytkownika i hasła. Nazwę hosta można dopisać do nazwy użytkownika (na przykład `scott@orcl`).

Wykonywanie instrukcji SELECT za pomocą SQL*Plus

Po zalogowaniu się do bazy danych za pomocą SQL*Plus można od razu uruchomić następujące polecenie `SELECT`, które zwraca bieżącą datę:

```
SELECT SYSDATE FROM dual;
```



W tej książce instrukcje SQL wyróżnione **pogrubieniem** należy wpisywać samodzielnie. Nie trzeba wpisywać niewyróżnionych instrukcji.

`SYSDATE` jest wbudowaną funkcją bazy danych, zwracającą bieżącą datę, `dual` jest natomiast tabelą zawierającą jeden wiersz.Więcej na temat tabeli `dual` dowiesz się w kolejnym rozdziale.



Instrukcje SQL należy kończyć średnikiem (`;`).

Rysunek 1.2 przedstawia datę zwróconą jako wynik opisanej wyżej instrukcji `SELECT`.

W programie SQL*Plus można edytować ostatnią instrukcję SQL. W tym celu należy wpisać `EDIT`. Jest to przydatne, gdy popełnimy błąd lub chcemy wprowadzić zmianę do instrukcji. W systemie Windows po wpisaniu `EDIT` jest uruchamiany Notatnik, w którym edytuje się instrukcje SQL. Po zakończeniu pracy Notatnika i zapisaniu instrukcji SQL jest ona przesyłana z powrotem do SQL*Plus, gdzie można ją uruchomić ponownie, wpisując ukośnik (`/`). W systemie Linux lub Unix domyślnym edytorem jest `ed`. Aby zapisać zmiany w instrukcji i opuścić `ed`, należy wpisać `wq`.

Rysunek 1.2.
Wynik instrukcji
SELECT SYSDATE
FROM dual;

```
SQL> SELECT SYSDATE FROM dual;
SYSDATE
-----
14/06/30
SQL> -
```

Kwestia polonizacji

Aby uzyskać format daty zaprezentowany na rysunku 1.2, a także polskie komunikaty z bazy danych widoczne na zrzutach w dalszej części książki, w konsoli Windows 7 należy wcześniej odpowiednio ustawić wartość zmiennej środowiskowej NLS_LANG:

```
set NLS_LANG=POLISH POLAND.EE8MSWIN1250
```

oraz stronę kodową konsoli:

```
chcp 1250
```

a dopiero potem uruchomić program sqlplus.

Aby polskie znaki w wynikach zapytań wyświetlanych w konsoli wyświetlały się poprawnie, należy we właściwościach okna konsoli zmienić czcionkę z Czcionki rastrowe na Lucida Console.

Rozwiązywanie problemu z błędem przy próbie skorzystania z edycji

Jeśli przy próbie edycji instrukcji w Windows pojawi się błąd SP2-0110, należy uruchomić SQL*Plus z uprawnieniami administratora. W Windows 7 można to zrobić, klikając prawym klawiszem myszy skrót do SQL*Plus i wybierając Uruchom jako administrator. Można ustawić to na stałe — aby to zrobić, należy kliknąć prawym klawiszem myszy skrót do SQL*Plus, wybrać Właściwości, a następnie w zakładce Zgodność zaznaczyć Uruchom ten program jako administrator.

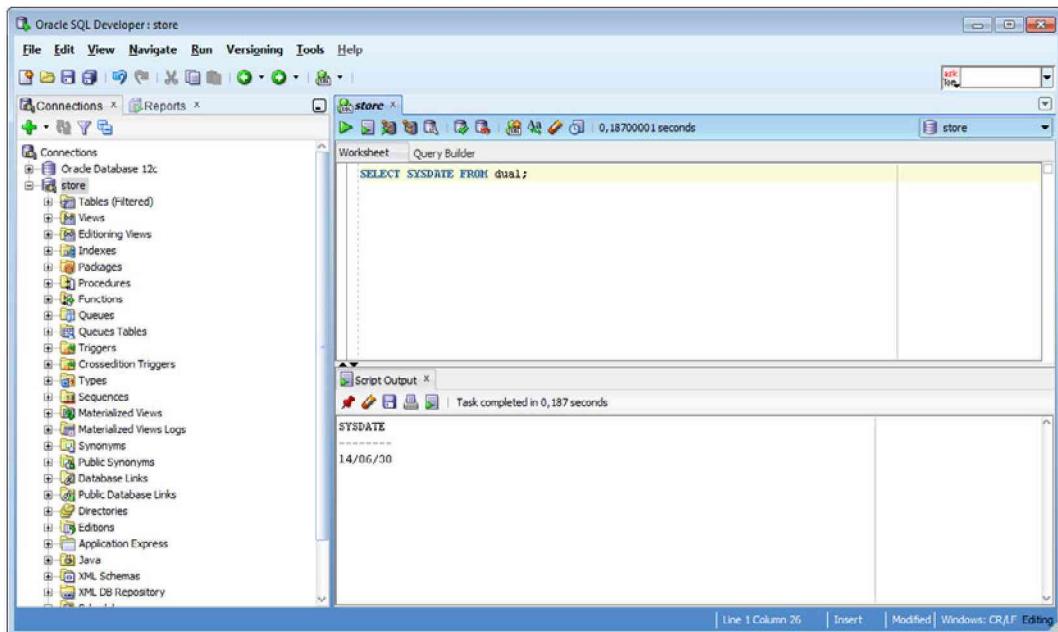
Möżesz też ustawić katalog, w którym SQL*Plus ma się uruchamiać. Aby to zrobić, należy kliknąć prawym klawiszem myszy skrót do SQL*Plus, wybrać Właściwości, a następnie zmienić katalog w znajdującym się w zakładce Skróty polu Rozpocznij w:. SQL*Plus będzie używał tego domyślnego katalogu do zapisywania i wczytywania plików. Möżesz na przykład ustawić ten katalog na C:\Moje_pliki_SQL i wtedy SQL*Plus będzie domyślnie zapisywał i wczytywał pliki z tego katalogu.

W SQL*Plus w systemie Windows można wracać do uruchamianych wcześniej poleceń, wciskając na klawiaturze klawisze ze strzałkami w góre i w dół.

Więcej informacji na temat edytowania instrukcji SQL w programie SQL*Plus znajduje się w rozdziale 3.

SQL Developer

Instrukcje SQL można również wprowadzać za pomocą programu SQL Developer. Ma on graficzny interfejs użytkownika, w którym można wpisywać instrukcje SQL, przeglądać tabele bazy danych, uruchamiać skrypty, edytować i debugować kod PL/SQL i wykonywać inne zadania. Może łączyć się z bazą danych Oracle w wersji 9.2.0.1 lub wyższej i jest dostępny dla wielu systemów operacyjnych. Rysunek 1.3 przedstawia okno programu SQL Developer.

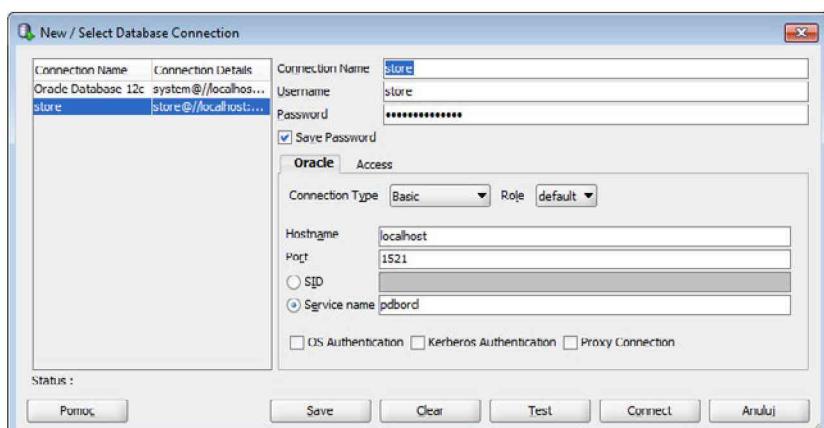


Rysunek 1.3. Okno programu SQL Developer, uruchomionego w systemie Windows

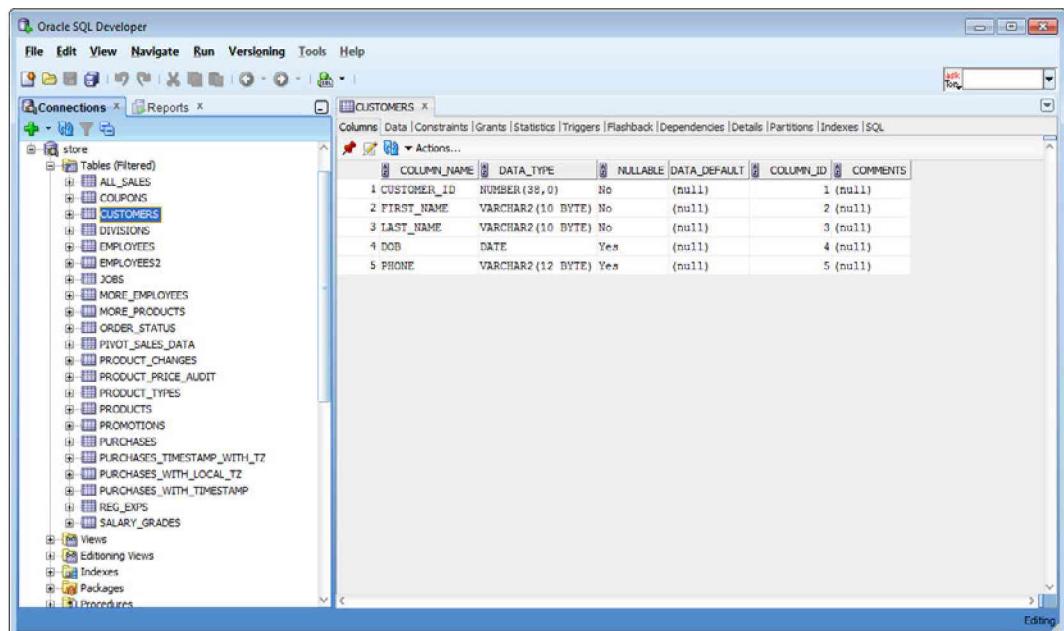
Należy pobrać wersję programu SQL Developer zawierającą Java Software Development Kit (SDK) lub mieć wcześniej zainstalowaną poprawną wersję Java SDK. Wymagana wersja Java SDK zależy od wersji programu SQL Developer. Szczegóły można sprawdzić na stronie programu SQL Developer w serwisie www.oracle.com.

Po poprawnym uruchomieniu programu SQL Developer niezbędne będzie utworzenie połączenia z bazą danych poprzez kliknięcie prawym klawiszem myszy *Connections* i wybranie *New Connection*. SQL Developer wyświetli okno dialogowe w którym należy wpisać szczegóły połączenia z bazą danych. Rysunek 1.4 pokazuje przykładowe okno dialogowe z wpisanyymi szczegółami połączenia.

Rysunek 1.4.
Konfigurowanie
połączenia z bazą
danych w programie
SQL Developer

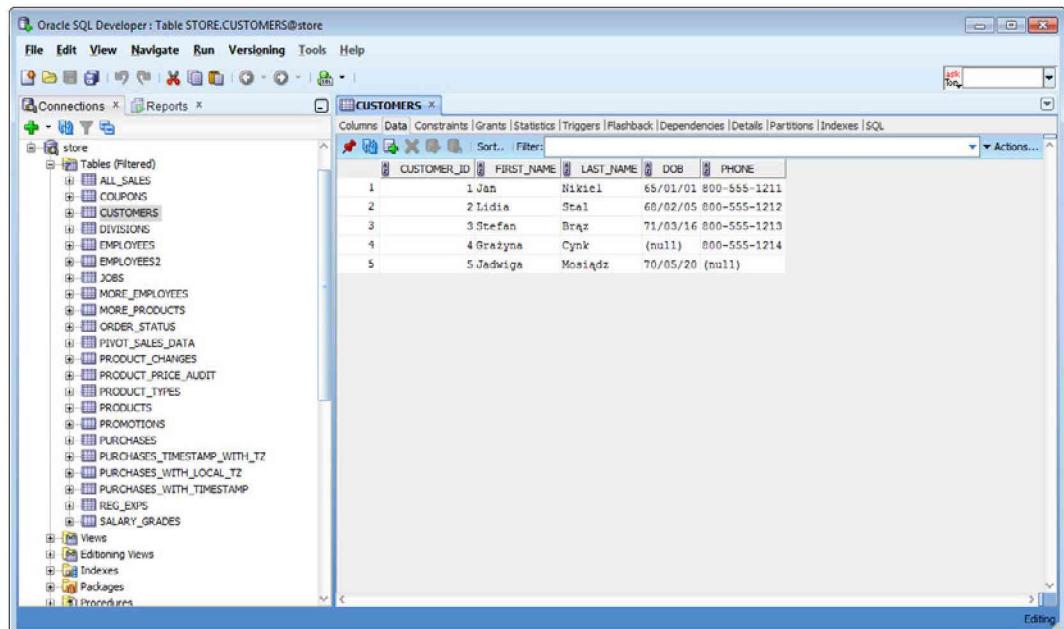


Po utworzeniu i przetestowaniu połączenia można używać programu SQL Developer do przeglądania tabel bazy danych i uruchamiania instrukcji. Rysunek 1.5 przedstawia szczegółowe informacje o tabellej customers.



Rysunek 1.5. Informacje o tabeli customers

Można również przeglądać dane przechowywane w tabeli po wybraniu zakładki *Data*, co pokazano na rysunku 1.6.



Rysunek 1.6. Dane przechowywane w tabeli customers

Z następnego podrozdziału dowiesz się, jak utworzyć schemat store wykorzystywany w dalszej części książki.

Tworzenie schematu bazy danych sklepu

W ofercie sklepu znajdują się książki, kasety wideo, płyty DVD i CD. W bazie danych będą przechowywane informacje o klientach, pracownikach, towarach i sprzedaży. Skrypt tworzący bazę danych *store_schema.sql*, przeznaczony dla programu SQL*Plus znajduje się w podkatalogu *SQL*, w katalogu, w którym rozpakowano plik z materiałami do książki. Skrypt *store_schema.sql* zawiera instrukcje DDL i DML używane do utworzenia schematu.

Zawartość skryptu

Otwórz skrypt w edytorze i przeanalizuj zawarte tam instrukcje. Ten podrozdział wprowadza instrukcje skryptu i pokazuje, jakie zmiany prawdopodobnie będziesz musiał do niego wprowadzić. Więcej na temat instrukcji zawartych w skrypcie dowiesz się w dalszej części tego rozdziału.

Usuwanie i tworzenie użytkownika

Pierwsza wykonywana instrukcja w skrypcie *store_schema.sql* to:

```
DROP USER store CASCADE;
```

Instrukcja `DROP USER` jest tutaj umieszczona, abyś nie musiał ręcznie usuwać użytkownika `store`, jeśli będziesz odtwarzać schemat w dalszej części książki.

Następna instrukcja tworzy użytkownika `store` z hasłem `store_password`:

```
CREATE USER store IDENTIFIED BY store_password;
```

Kolejna instrukcja umożliwia użytkownikowi `store` połączenie z bazą danych i tworzenie elementów bazy danych:

```
GRANT connect, resource TO store;
```

Przydzielanie miejsca na tabelę

Kolejna instrukcja przydziela użytkownikowi `store` 10 MB w przestrzeni tabel `users`:

```
ALTER USER store QUOTA 10M ON users;
```

Przestrzenie tabel są używane przez bazę danych do przechowywania tabel i innych elementów bazy danych. Więcej na temat przestrzeni tabel dowiesz się w rozdziale 10. Większość baz danych ma przestrzeń tabel `users` do przechowywania danych użytkowników. Aby to sprawdzić, najpierw połącz się z bazą danych jako użytkownik uprzywilejowany (na przykład użytkownik `system`) i uruchom poniższą instrukcję:

```
SELECT property_value
FROM database_properties
WHERE property_name = 'DEFAULT_PERMANENT_TABLESPACE';
```

PROPERTY_VALUE

USERS

Ta instrukcja zwraca nazwę przestrzeni tabel, której należy użyć w instrukcji `ALTER USER`. W mojej bazie danych jest to przestrzeń `users`.

Jeśli nazwa przestrzeni tabel zwrócona przez tę instrukcję jest inna, musisz zamienić `users` w instrukcji `ALTER USER` na zwroconą przez powyższą instrukcję. Jeśli na przykład nazwa Twojej przestrzeni tabel to `another_ts`, zmień instrukcję w skrypcie na:

```
ALTER USER store QUOTA 10M ON another_ts;
```

Ustanowienie połączenia

Poniższa instrukcja skryptu tworzy połączenie, korzystając z danych użytkownika `user`:

```
CONNECT store/store_password;
```

Tę instrukcję będziesz musiał zmodyfikować, jeśli baza danych, z którą się łączysz, znajduje się na innym komputerze. Na przykład jeśli łączysz się z bazą o nazwie `orcl`, zmień instrukcję `CONNECT` na:

```
CONNECT store/store_password@orcl;
```

Użycie dołączanych baz danych

Dołączane bazy danych (*pluggable databases*) to nowa funkcjonalność w Oracle Database 12c. Są one twozone w zewnętrznym kontenerze bazy danych. Pozwalają one oszczędzać zasoby systemowe, upraszczają administrację i są zazwyczaj tworzone przez administratorów baz danych, dlatego dokładne omówienie dołączanych baz danych wykracza poza zakres tej książki.

Jeżeli korzystasz z tej funkcjonalności, będziesz musiał zmodyfikować instrukcję `CONNECT` w skrypcie i dodać nazwę dołączanej bazy danych. Jeśli na przykład nazwą Twojej dołączanej bazy danych jest `pdborcl`, należy instrukcję w skrypcie zmienić na:

```
CONNECT store/store_password@//localhost/pdborcl;
```

Jeśli wprowadziłeś jakieś zmiany do skryptu `store_schema.sql`, zapisz zmodyfikowaną wersję.

Pozostałe instrukcje w skrypcie tworzą tabele i inne elementy niezbędne do stworzenia przykładowego sklepu. Dowiesz się więcej na temat tych instrukcji w dalszej części tego rozdziału.

Uruchamianie skryptu

Aby utworzyć schemat `store`, należy:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik mający uprawnienia do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty jako użytkownik `system`, który posiada wszystkie wymagane uprawnienia.
3. Jeśli używasz dołączanych baz danych, należy wybrać dołączaną bazę danych jako kontener sesji. Będzie to możliwe po uruchomieniu SQL*Plus z uprawnieniami `sysdba`. Jeśli na przykład nazwa dołączanej bazy danych to `pdborcl`, w konsoli wykonaj poniższe instrukcje:

```
sqlplus "/as sysdba"  
ALTER SESSION SET CONTAINER=pdborcl;
```

Przed rozpoczęciem pracy z dołączaną bazą danych może okazać się też konieczne jej otwarcie instrukcją:

```
alter database open;
```

4. Uruchomić skrypt `store_schema.sql` z programu SQL*Plus, korzystając z polecenia `@`. Polecenie `@` ma następującą składnię:

```
@ katalog\store_schema.sql
```

gdzie `katalog` oznacza katalog, w którym znajduje się skrypt `store_schema.sql`.

Jeżeli skrypt znajduje się w `E:\sql_book\SQL`, należy wpisać:

```
@ E:\sql_book\SQL\store_schema.sql
```

Jeżeli skrypt `store_schema.sql` znajduje się w katalogu, którego nazwa zawiera spację, należy umieścić całą ścieżkę dostępu w cudzysłowie, za poleceniem `@`. Na przykład:

```
@ "E:\Oracle SQL book\sql_book\SQL\store_schema.sql"
```

Jeżeli korzysta się z systemu Unix lub Linux i skrypt został zapisany w katalogu `SQL`, umieszczonym w katalogu `tmp`, należy wpisać:

```
@ /tmp/SQL/store_schema.sql
```



W ścieżkach dostępu do katalogów w systemie Windows jest wykorzystywany lewy ukośnik (`\`), a w systemach Unix i Linux — prawy (`/`).

Po zakończeniu pracy skryptu *store_schema.sql* pozostaniemy połączeni z bazą danych jako użytkownik *store*. Hasło tego użytkownika to *store_password*.

W dalszej części książki będzie konieczne uruchamianie innych skryptów. Przed wykonaniem każdego ze skryptów należy wykonać poniższe czynności:

- Jeśli baza danych nie ma przestrzeni tabel *users*, należy zmodyfikować instrukcję `ALTER USER` w skrypcie.
- Jeśli konieczne jest dodanie nazwy komputera, na którym znajduje się baza danych, należy zmodyfikować instrukcję `CONNECT` w skrypcie.
- Jeśli korzystasz z dołączanych baz danych, należy zmodyfikować instrukcję `CONNECT` oraz uruchomić instrukcję `ALTER SESSION SET CONTAINER` przed wykonaniem skryptu.

Nie musisz w tej chwili modyfikować wszystkich skryptów. Po prostu pamiętaj o tym przed uruchomieniem każdego z nich.

Instrukcje DDL używane do tworzenia schematu bazy danych sklepu

Instrukcje języka DDL są wykorzystywane do tworzenia kont użytkowników i tabel oraz wielu innych struktur w bazie danych. W tym podrozdziale poznamy instrukcje DDL użyte do utworzenia konta użytkownika *store* i niektórych tabel.



Instrukcje SQL prezentowane w tym rozdziale są zawarte w skrypcie *store_schema.sql*. Nie trzeba ich wpisywać samodzielnie — wystarczy jedynie uruchomić skrypt.

W kolejnych podrozdziałach opisano:

- tworzenie konta użytkownika bazy danych,
- typy danych najczęściej wykorzystywane w bazie danych Oracle,
- niektóre tabele tworzonej bazy danych.

Tworzenie konta użytkownika bazy danych

Do tworzenia konta użytkownika bazy danych służy instrukcja `CREATE USER`. Uproszczona składnia tego polecenia ma następującą postać:

`CREATE USER nazwa_użytkownika IDENTIFIED BY hasło;`

gdzie:

- *nazwa_użytkownika* jest nazwą użytkownika,
- *hasło* jest hasłem dostępu tego użytkownika.

Na przykład poniższa instrukcja `CREATE USER` tworzy konto użytkownika *store* z hasłem *store_password*:

`CREATE USER store IDENTIFIED BY store_password;`

Jeżeli chcemy, aby użytkownik mógł pracować z bazą danych, musimy nadać mu odpowiednie **uprawnienia**. W przypadku *store* użytkownik musi mieć możliwość zalogowania się do bazy danych (co wymaga uprawnienia *connect*) oraz tworzenia tabel (co wymaga uprawnienia *resource*). Uprawnienia są przydzielane przez uprzywilejowanego użytkownika (na przykład użytkownika *system*) za pomocą instrukcji `GRANT`.

Przedstawiona poniżej instrukcja powoduje przydelenie uprawnień *connect* i *resource* użytkownikowi *store*:

`GRANT connect, resource TO store;`

W wielu przykładach w tej książce jest wykorzystywany schemat *store*. Zanim szczegółowo zajmiemy się tabelami bazy danych sklepu, musisz poznać typy danych najczęściej używane w bazach Oracle.

Typy danych często używane w bazach Oracle

Najczęściej wykorzystywane w bazach Oracle typy danych zostały opisane w tabeli 1.1. Wszystkie typy danych zostały opisane w dodatku.

Tabela 1.1. Typy danych często wykorzystywane w bazach Oracle

Typ Oracle	Znaczenie
CHAR(<i>długość</i>)	Przechowuje wpisy o stałej długości. Jeżeli wprowadza się wpis, który jest krótszy, niż określa to parametr <i>długość</i> , jest on wypełniany spacjami. Do przechowywania dwuznakowych wpisów o stałej długości można na przykład wykorzystać typ CHAR(2). Jeżeli zapiszemy w nim C, na końcu zostanie wstawiony jeden znak spacji. Natomiast CA zostanie zapisane w niezmienionej postaci
VARCHAR2(<i>długość</i>)	Przechowuje wpisy o zmiennej długości. Parametr <i>długość</i> określa maksymalną długość wpisu. Typ VARCHAR2(20) może na przykład zostać użyty do przechowywania wpisu o długości nieprzekraczającej 20 znaków. Jeżeli zostanie wprowadzony krótszy wpis, nie będzie on dopełniany znakami spacji
DATE	Przechowuje datę i czas. Typ DATE przechowuje wiek, wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minutę i sekundę. Typ DATE może zostać użyty do przechowywania dat i godzin między 1 stycznia 4712 p.n.e. i 31 grudnia 9999 n.e.
INTEGER	Przechowuje liczby całkowite, czyli takie jak 1, 10 czy 115. Liczby całkowite nie mają części ułamkowej.
NUMBER(<i>precyzja</i> , <i>skala</i>)	Przechowuje liczby zmienoprzecinkowe. Parametr <i>precyzja</i> określa maksymalną liczbę cyfr. Jest to łączna liczba cyfr zapisanych po lewej i po prawej stronie separatora dziesiętnego. Baza danych Oracle obsługuje maksymalnie precyzję 38. Parametr <i>skala</i> określa maksymalną liczbę cyfr po prawej stronie separatora dziesiętnego. Jeżeli nie zostanie podany żaden z parametrów, można zapisać dowolną liczbę z precyzją do 38. Każda próba zapisu liczby przekraczającej parametr <i>precyzja</i> jest odrzucana przez bazę danych

Poniżej przedstawiono kilka przykładów przechowywania w bazie danych liczb o typie NUMBER:

Format	Wprowadzona liczba	Zapisana liczba
NUMBER	1234,567	1234,567
NUMBER(6, 2)	123,4567	123,46
NUMBER(6, 2)	12345,67	Liczba przekraczająca określona precyzję i w związku z tym jest odrzucana przez bazę danych

Tabele w schemacie bazy danych store

Z tego podrozdziału dowiesz się, jak tworzy się tabele w schemacie bazy danych store. Oto niektóre przechowywane w niej informacje:

- dane klientów,
- rodzaj sprzedawanych produktów,
- opis produktów,
- dane o towarze kupionym przez klientów,
- dane pracowników sklepu,
- wysokość wynagrodzeń.

Informacje są przechowywane w następujących tabelach:

- **customers** przechowuje informacje o klientach,
- **product_types** przechowuje informacje o produktach sprzedawanych w sklepie,

- **products** przechowuje szczegółowe informacje o produktach,
- **purchases** przechowuje informacje o tym, które produkty zostały kupione przez danych klientów,
- **employees** przechowuje informacje o pracownikówach,
- **salary_grades** przechowuje informacje o wynagrodzeniach.

W kolejnych podrozdziałach zostaną opisane szczegółowo kilka tabel, a także instrukcje CREATE TABLE, użyte do utworzenia tych elementów. Instrukcje te znajdują się w skrypcie *store_schema.sql*.

Tabela customers

W tabeli *customers* są zapisywane dane klientów:

- imię,
- nazwisko,
- data urodzenia (*dob*),
- numer telefonu.

Każdy z tych elementów musi być umieszczony w osobnej kolumnie. W skrypcie *store_schema.sql* tabela *customers* jest tworzona za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE customers (
    customer_id INTEGER CONSTRAINT customers_pk PRIMARY KEY,
    first_name VARCHAR2(10) NOT NULL,
    last_name VARCHAR2(10) NOT NULL,
    dob DATE,
    phone VARCHAR2(12)
);
```

Jak widzimy, tabela *customers* zawiera pięć kolumn — po jednej dla każdego elementu z listy oraz dodatkową o nazwie *customer_id*:

- **customer_id** zawiera unikatową liczbę całkowitą dla każdego wiersza tabeli. Wszystkie tabele powinny zawierać jedną lub kilka kolumn, umożliwiających jednoznaczną identyfikację wierszy. Takie kolumny nazywamy **kluczami głównymi**. Klauzula CONSTRAINT wskazuje, że kolumna *customer_id* jest kluczem głównym, i ogranicza wartości przechowywane w niej. W przypadku kolumny *customer_id* słowa kluczowe PRIMARY KEY wskazują, że każdy wiersz musi zawierać unikatową wartość. Można również dołączyć opcjonalną nazwę ograniczenia, którą należy wpisać tuż za słowem kluczowym CONSTRAINT, na przykład *customers_pk*. Należy zawsze nazywać ograniczenia (więzy) kluczy głównych. Ułatwia to znalezienie przyczyn ewentualnych błędów.
- **first_name** zawiera imię klienta. Zastosowanie w niej więzów NOT NULL oznacza, że przy dodawaniu lub modyfikowaniu wiersza musi zostać podana jakaś wartość. Jeżeli to ograniczenie zostanie pominięte, użytkownik nie musi podawać wartości, a kolumna może pozostać pusta.
- **last_name** zawiera nazwisko klienta. Ta kolumna również ma więzy NOT NULL, co oznacza, że przy modyfikowaniu lub dodawaniu wiersza konieczne jest podanie wartości.
- **dob** zawiera datę urodzenia klienta. Nie określono dla niej więzów NOT NULL, więc domyślną wartością jest NULL. Podczas dodawania wiersza podanie wartości nie jest wymagane.
- **phone** zawiera numer telefonu klienta. W tej kolumnie również nie ma więzów NOT NULL.

Skrypt *store_schema.sql* umieszcza w tabeli *customers* następujące wiersze:

customer_id	first_name	last_name	dob	phone
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stał	68/02/05	800-555-1212
3	Stefan	Braż	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Należy zwrócić uwagę, że data urodzenia czwartego klienta nie została wpisana, podobnie jak numer telefonu piątego klienta.

Aby przejrzeć wiersze tabeli *customers*, należy uruchomić w SQL*Plus następującą instrukcję SELECT:

```
SELECT * FROM customers;
```

Gwiazdka (*) oznacza, że chcemy pobrać wszystkie kolumny z tabeli *customers*.

Tabela product_types

W tabeli *product_types* są przechowywane nazwy rodzajów produktów sprzedawanych w sklepie. Została ona utworzona przez skrypt *store_schema.sql* za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE product_types (
    product_type_id INTEGER CONSTRAINT product_types_pk PRIMARY KEY,
    name VARCHAR2(10) NOT NULL
);
```

Tabela *product_types* zawiera dwie kolumny:

- **product_type_id** jednoznacznie identyfikuje każdy wiersz w tabeli i jest jej kluczem głównym. Każdy wiersz w tabeli *product_types* musi posiadać niepowtarzaną wartość (liczbę całkowitą) w tej kolumnie.
- **name** zawiera nazwę rodzaju produktu i ma więcej NOT NULL, więc przy dodawaniu wiersza trzeba podać określoną w niej wartość.

Skrypt *store_schema.sql* umieszcza w tabeli *product_types* następujące wiersze:

product_type_id	name
1	Książka
2	VHS
3	DVD
4	CD
5	Czasopismo

Tabela *product_types* zawiera informacje o rodzajach produktów sprzedawanych w sklepie. Każdy musi należeć do jednej z tych kategorii.

Aby przejrzeć wiersze tabeli *product_types*, należy uruchomić w SQL*Plus następującą instrukcję SELECT:

```
SELECT * FROM product_types;
```

Tabela products

W tabeli *products* są przechowywane następujące dane na temat towarów dostępnych w sklepie:

- typ,
- nazwa,
- opis,
- cena.

W skrypcie *store_schema.sql* tabela *products* jest tworzona za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE products (
    product_id INTEGER CONSTRAINT products_pk PRIMARY KEY,
    product_type_id INTEGER
        CONSTRAINT products_fk_product_types
        REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);
```

W tej tabeli znajdują się następujące kolumny:

- **product_id** jednoznacznie identyfikuje każdy wiersz w tabeli i jest jej kluczem głównym.
- **product_type_id** przypisuje każdy produkt do określonego typu i jest odwołaniem do kolumny *product_type_id* w tabeli *product_types*. Jest to **klucz obcy**, ponieważ odsyła do kolumny w innej tabeli. Tabelę zawierającą klucz obcy (w tym przypadku tabelę *products*) nazywamy tabelą **podzieloną**,

a tabelę, którą wskazuje odwołanie (w tym przypadku `product_types`) — tabelą **nadrzędną**. Taki typ relacji nazywamy relacją **nadrzędną/podrzędną**. Gdy dodajemy nowy produkt, tworzymy po-wiązanie z typem produktu przez wprowadzenie odpowiedniej wartości `product_types.product_type_id` do kolumny `products.product_type_id` (zostanie to później przedstawione na przykładzie).

- **name** zawiera nazwę produktu. Ma więcej NOT NULL.
- **description** zawiera opcjonalny opis produktu.
- **price** zawiera opcjonalną cenę produktu. Ta kolumna jest definiowana jako NUMBER(5, 2) — pre-cyza wynosi 5, można więc wprowadzić maksymalnie 5 cyfr. Skala wynosi 2, co oznacza, że tylko 2 z tych 5 cyfr mogą znajdować się po prawej stronie separatora dziesiętnego.

Poniżej przedstawiono kilka wierszy przechowywanych w tabeli `products`:

product_id	product_type_id	name	description	price
1	1	Nauka współczesnej nauki	Opis współczesnej nauki	19,95
2	1	Chemia	Wprowadzenie do chemii	30
3	2	Supernowa	Eksplozja gwiazdy	25,99
4	2	Wojny czołgów	Film akcji o nadchodzącej wojnie	13,95

Wartość `product_type_id` pierwszego wiersza w tabeli `products` wynosi 1, co oznacza, że jest to książka (wartość `product_type_id` odpowiada typowi produktu „książka” w tabeli `product_types`). Drugi produkt również jest książką, trzeci i czwarty natomiast to kasety wideo (wartość w kolumnie `product_type_id` dla tych produktów wynosi 2, co odpowiada typowi produktu „VHS” w tabeli `product_types`).

Aby przejrzeć wiersze tabeli `products`, należy uruchomić w SQL*Plus następującą instrukcję SELECT:

```
SELECT * FROM products;
```

Tabela purchases

W tabeli `purchases` są przechowywane informacje o zakupach dokonanych przez poszczególnych klientów. Przechowywane są następujące informacje o każdym zakupie:

- identyfikator kupionego produktu,
- identyfikator klienta,
- liczba jednostek produktów nabitych przez klienta.

W skrypcie `store_schema.sql` tabela `purchases` jest tworzona za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE purchases (
    product_id INTEGER
        CONSTRAINT purchases_fk_products
        REFERENCES products(product_id),
    customer_id INTEGER
        CONSTRAINT purchases_fk_customers
        REFERENCES customers(customer_id),
    quantity INTEGER NOT NULL,
    CONSTRAINT purchases_pk PRIMARY KEY (product_id, customer_id)
);
```

W tabeli znajdują się następujące kolumny:

- **product_id**, która zawiera identyfikator nabitego produktu. Wartość musi być zgodna z wartością w kolumnie `product_id` tabeli `products`.
- **customer_id**, zawierająca identyfikator klienta, który dokonał zakupu. Wartość musi być zgodna z wartością `customer_id` tabeli `customers`.
- **quantity** z liczbą jednostek produktu nabitych przez klienta.

Tabela purchases posiada więcej klucza głównego o nazwie purchases_pk, które obejmuje dwie kolumny: product_id i customer_id. Połączenie wartości z tych dwóch kolumn musi być unikatowe dla każdego wiersza. Jeżeli klucz główny składa się z kilku kolumn, nazywamy go **złożonym kluczem głównym**.

Poniżej przedstawiono kilka wierszy przechowywanych w tabeli purchases:

product_id	customer_id	quantity
1	1	1
2	1	3
1	4	1
2	2	1
1	3	1

Jak widać, kombinacja wartości w kolumnach product_id i customer_id nie powtarza się w żadnym z wierszy.

Aby przejrzeć wiersze tabeli purchases, należy uruchomić w SQL*Plus następującą instrukcję SELECT:

```
SELECT * FROM purchases;
```

Tabela employees

W tabeli employees są przechowywane informacje o pracownikach. Znajdują się w niej następujące dane:

- identyfikator pracownika,
- identyfikator przełożonego pracownika,
- imię,
- nazwisko,
- stanowisko,
- wynagrodzenie.

W skrypcie *store_schema.sql* tabela employees jest tworzona za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE employees (
    employee_id INTEGER
        CONSTRAINT employees_pk PRIMARY KEY,
    manager_id INTEGER,
    first_name VARCHAR2(10) NOT NULL,
    last_name VARCHAR2(10) NOT NULL,
    title VARCHAR2(20),
    salary NUMBER(6, 0)
);
```

Skrypt *store_schema.sql* umieszcza w tabeli employees następujące wiersze:

employee_id	manager_id	first_name	last_name	title	salary
1	Jan	Kowalski	CEO	800000	
2	1	Roman	Joświerz	Kierownik sprzedaży	600000
3	2	Fryderyk	Helc	Sprzedawca	150000
4	2	Zofia	Nowak	Sprzedawca	500000

Jan Kowalski nie ma kierownika, ponieważ jest właścicielem sklepu.

Tabela salary_grades

W tabeli salary_grades są przechowywane dane o przedziałach wynagrodzeń pracowników:

- identyfikator przedziału wynagrodzeń,
- dolna granica przedziału wynagrodzeń,
- górna granica przedziału wynagrodzeń.

W skrypcie *store_schema.sql* tabela salary_grades jest tworzona za pomocą następującej instrukcji CREATE TABLE:

```
CREATE TABLE salary_grades (
    salary_grade_id INTEGER CONSTRAINT salary_grade_pk PRIMARY KEY,
    low_salary NUMBER(6, 0),
    high_salary NUMBER(6, 0)
);
```

Skrypt *store_schema.sql* umieszcza w tabeli *salary_grades* następujące wiersze:

salary_grade_id	low_salary	high_salary
1	1	250000
2	250001	500000
3	500001	750000
4	750001	999999

Dodawanie, modyfikowanie i usuwanie wierszy

Z tego podrozdziału dowiesz się, jak dodawać, modyfikować i usuwać wiersze z tabel bazy danych, korzystając z instrukcji `INSERT`, `UPDATE` i `DELETE`. Zmiany wprowadzone do wierszy można zapisać na stałe za pomocą instrukcji `COMMIT` lub cofnąć, używając polecenia `ROLLBACK`.Więcej informacji na ten temat znajduje się w rozdziale 9.

Dodawanie wiersza do tabeli

Do wstawiania nowych wierszy do tabeli służy instrukcja `INSERT`. Możemy w niej określić następujące informacje:

- tabelę, do której będzie wstawiany wiersz,
- listę kolumn, dla których chcemy określić wartości,
- listę wartości, które mają zostać zapisane w określonych kolumnach.

Przy wstawianiu wiersza należy podać wartość klucza głównego i wszystkich innych kolumn zdefiniowanych jako `NOT NULL`. Nie jest konieczne wpisywanie wartości dla innych kolumn — zostanie im domyślnie przypisana wartość `NULL`.

Aby dowiedzieć się, które kolumny są zdefiniowane jako `NOT NULL`, możemy użyć polecenia SQL*Plus `DESCRIBE`. Poniżej wyświetlono za jego pomocą informacje o tabeli *customers*:

```
SQL> DESCRIBE customers;
Name          Null?    Type
-----        -----
CUSTOMER_ID   NOT NULL NUMBER(38)
FIRST_NAME    NOT NULL VARCHAR2(10)
LAST_NAME     NOT NULL VARCHAR2(10)
DOB           DATE
PHONE         VARCHAR2(12)
```

Kolumny *customer_id*, *first_name* i *last_name* są zdefiniowane z wiezami `NOT NULL`, należy więc podać dla nich wartość. Kolumny *dob* i *phone* nie wymagają wpisywania wartości, można je zatem pominąć, ponieważ automatycznie zostanie im przypisana wartość `NULL`.

Możemy teraz uruchomić poniższą instrukcję `INSERT`, która wstawi wiersz do tabeli *customers*. Należy zwrócić uwagę, że kolejność wartości na liście `VALUES` jest zgodna z tą, w jakiej zostały wymienione kolumny:

```
SQL> INSERT INTO customers (
  2   customer_id, first_name, last_name, dob, phone
  3 ) VALUES (
  4   6, 'Fryderyk', 'Nikiel', '1970/01/01', '800-555-1215'
  5 );
```

1 row created.



SQL*Plus automatycznie numeruje wiersze po naciśnięciu klawisza *Enter* na końcu wiersza.

W przedstawionym przykładzie SQL*Plus informuje, że po wykonaniu instrukcji `INSERT` został wstawiony jeden wiersz. Można to sprawdzić, uruchamiając następującą instrukcję `SELECT`:

```
SELECT *
FROM customers;
```

customer_id	first_name	last_name	dob	phone
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Braż	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
6	Fryderyk	Nikiel	70/01/01	800-555-1215

Nowy wiersz został umieszczony na końcu tabeli.

Domyślnie baza danych Oracle wyświetla daty w formacie RR/MM/DD, gdzie RR oznacza dwie ostatnie cyfry roku, MM — dwie cyfry określające miesiąc, a DD — dzień. W bazie danych tak naprawdę przechowywane są wszystkie cztery cyfry roku, domyślnie jednak są wyświetlane tylko dwie ostatnie.

Przy wstawianiu wiersza do tabeli `customers` należy podać unikatową wartość dla kolumny `customer_id`. Baza danych Oracle uniemożliwi wstawienie wiersza z wartością klucza głównego, która została już wykorzystana w tabeli. Poniższa instrukcja `INSERT` spowoduje błąd, ponieważ wiersz z `customer_id` równym jeden już istnieje:

```
SQL> INSERT INTO customers (
 2   customer_id, first_name, last_name, dob, phone
 3 ) VALUES (
 4   1, 'Eliza', 'Kowal', '1971/01/02', '800-555-1225'
 5 );
```



```
INSERT INTO customers (
*
BŁĄD w linii 1:
ORA-00001: naruszono więzy unikatowe (STORE.CUSTOMERS_PK)
```

Należy zwrócić uwagę, że w komunikacie o błędzie jest wyświetlana nazwa więzów (`CUSTOMERS_PK`). Z tego właśnie powodu należy zawsze nadawać nazwę więzom klucza głównego, w przeciwnym bowiem razie baza danych przypisze im taką, która będzie niezrozumiała, na przykład `SYS_C0011277`.

Modyfikowanie istniejącego wiersza w tabeli

Do zmieniania zawartości wierszy w tabeli służy instrukcja `UPDATE`. Używając jej, zwykle podajemy następujące informacje:

- nazwę tabeli, w której będą zmieniane wiersze,
- klauzulę `WHERE`, określającą, które wiersze będą zmieniane,
- listę nazw kolumn i ich nowych wartości, określana za pomocą klauzuli `SET`.

Za pomocą instrukcji `UPDATE` można zmienić zawartość jednego lub kilku wierszy. Jeżeli określonych zostanie kilka wierszy, ta sama zmiana zostanie wprowadzona w każdym z nich. W poniższym przykładzie zmieniono wartość `last_name` dla drugiego klienta na „Żelazo”:

```
UPDATE customers
SET last_name = 'Żelazo'
WHERE customer_id = 2;
```

1 row updated.

SQL*Plus potwierdzi, że został zaktualizowany jeden wiersz.



Jeżeli nie zostanie określona klauzula WHERE, aktualizacja obejmie wszystkie wiersze.

Poniższe zapytanie pozwala upewnić się, że aktualizacja przebiegła poprawnie:

```
SELECT *
FROM customers
WHERE customer_id = 2;

customer_id first_name last_name    dob      phone
-----
2          Lidia       Źelazo   68/02/05 800-555-1212
```

Usuwanie wiersza z tabeli

Do usuwania wierszy z tabeli służy polecenie DELETE. Najczęściej stosujemy z tą instrukcją klauzulę WHERE, aby ograniczyć liczbę usuwanych wierszy. Jeżeli nie określmy tej wartości, z tabeli zostaną usunięte *wszystkie* wiersze.

Poniższa instrukcja DELETE usuwa informacje o drugim kliencie:

```
DELETE FROM customers
WHERE customer_id = 2;
```

1 row deleted.

Do cofnięcia zmian wprowadzonych w wierszach służy instrukcja ROLLBACK:

```
ROLLBACK;
```

Rollback complete.



Za pomocą instrukcji COMMIT można utrważyć zmiany wprowadzone w wierszach.
Więcej informacji na ten temat znajduje się w rozdziale 9.

Łączenie z bazą danych i rozłączanie

Gdy jesteś połączony z bazą danych, SQL*Plus utrzymuje aktywną sesję bazy danych. Gdy rozłączasz się z bazą danych, sesja jestkończona. Można rozłączyć się z bazą danych bez zamknięcia SQL*Plus za pomocą instrukcji DISCONNECT:

```
DISCONNECT
```

Domyślnie przy rozłączaniu automatycznie wykonywana jest instrukcja COMMIT.

Można ponownie połączyć się z bazą danych, wpisując CONNECT. Aby ponownie połączyć się ze schematem store, należy wpisać store jako nazwę użytkownika i store_password jako hasło:

```
CONNECT store/store_password
```

Kończenie pracy SQL*Plus

Do kończenia pracy programu SQL*Plus służy polecenie EXIT:

```
EXIT
```

Jeżeli praca programu SQL*Plus zostanie zakończona w ten sposób, program automatycznie wyda instrukcję COMMIT. Jeżeli SQL*Plus zostanie zamknięty nieprawidłowo (na przykład zawiesi się komputer, na którym jest uruchomiony), zostanie wykonana automatycznie instrukcja ROLLBACK. Więcej informacji na ten temat znajduje się w rozdziale 9.

Wprowadzenie do Oracle PL/SQL

PL/SQL jest językiem proceduralnym, opracowanym przez Oracle, który umożliwia dodawanie konstrukcji programistycznych do instrukcji SQL. Jest wykorzystywany głównie do tworzenia procedur i funkcji w bazach danych zawierających logikę biznesową. Zawiera standardowe elementy programistyczne, takie jak:

- deklaracje zmiennych,
- logikę warunkową (`if-then-else` itd.),
- pętle,
- procedury i funkcje.

Poniższa instrukcja `CREATE PROCEDURE` tworzy procedurę o nazwie `update_product_price()`. Mnoży ona cenę produktu razy mnożnik — identyfikator produktu oraz mnożnik są przesyłane do procedury jako parametry. Jeżeli określony produkt nie istnieje, procedura nie wykonuje żadnych działań, w przeciwnym razie aktualizuje cenę produktu.



Uwaga Nie należy na razie analizować szczegółów prezentowanego kodu PL/SQL — zostaną one opisane w rozdziale 12.

```
CREATE PROCEDURE update_product_price (
    p_product_id IN products.product_id%TYPE,
    p_factor IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- policz produkty z przesłanym product_id
    -- (będzie 1, jeżeli produkt istnieje)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- jeżeli produkt istnieje (czyli product_count = 1),
    -- aktualizuj cenę tego produktu
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
/
```

Wyjątki są używane do obsługi błędów występujących w kodzie PL/SQL. Blok `EXCEPTION` w powyższym przykładzie wykonuje instrukcję `ROLLBACK`, jeżeli kod zgłosi wyjątek.

Podsumowanie

W tym rozdziale dowiedziałeś się o tym, że:

- relacyjna baza danych jest zbiorem powiązanych z sobą informacji, uporządkowanych w strukturę zwane tabelami,
- strukturalny język zapytań (SQL) jest standardowym językiem zaprojektowanym do uzyskiwania dostępu do baz danych,

42 Oracle Database 12c SQL

- SQL*Plus umożliwia wykonywanie instrukcji SQL i poleceń programu SQL*Plus,
- SQL Developer jest graficznym narzędziem służącym do tworzenia baz danych,
- PL/SQL jest językiem proceduralnym, opracowanym przez Oracle, zawierającym instrukcje programistyczne.

Następny rozdział zawiera informacje o pobieraniu danych z tabel bazy danych.

ROZDZIAŁ

2

Pobieranie informacji z tabel bazy danych

Z tego rozdziału dowiesz się, jak:

- pobierać informacje z jednej lub kilku tabel bazy danych za pomocą instrukcji SELECT,
- wykonywać obliczenia za pomocą wyrażeń arytmetycznych,
- pobierać jedynie interesujące Cię wiersze, stosując klauzulę WHERE,
- sortować wiersze pobrane z tabeli.



Aby samodzielnie wykonywać prezentowane ćwiczenia, uzyskując takie same wyniki jak zaprezentowane w tym rozdziale, należy w tym miejscu wykonać ponownie skrypt `store_schema.sql`.

Wykonywanie instrukcji SELECT dla jednej tabeli

Instrukcja SELECT służy do pobierania informacji z tabel bazy danych. W najprostszej postaci tej instrukcji należy określić tabelę i kolumny, z których chcemy pobrać dane.

Poniższa instrukcja SELECT pobiera kolumny `customer_id`, `first_name`, `last_name`, `dob` i `phone` z tabeli `customers`.

```
SELECT customer_id, first_name, last_name, dob, phone  
FROM customers;
```

Tuż za słowem kluczowym SELECT należy wpisać nazwy kolumn, z których mają być pobierane dane. Za słowem kluczowym FROM należy wpisać nazwę tabeli. Instrukcję SQL trzeba zakończyć średnikiem (;).

System zarządzania bazą danych nie wymaga dokładnych informacji na temat tego, jak uzyskać dostęp do żądanych danych. Wystarczy jedynie, że określmy, czego chcemy, a szczegółami zajmie się oprogramowanie.

Po wpisaniu instrukcji SQL i naciśnięciu klawisza *Enter* zostanie ona wykonana, a wyniki zostaną zwrócone do SQL*Plus w celu wyświetlenia ich na ekranie:

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Wiersze zwrócone przez bazę danych nazywamy **zestawem wyników**. Zwróć uwagę na to, że:

- oprogramowanie Oracle zmienia litery nazw kolumn na wielkie;
- kolumny znakowe i z datami są wyrównywane do lewej;
- kolumny liczbowe są wyrównywane do prawej;
- po wybraniu polskich ustawań lokalnych daty są wyświetlane w formacie RR/MM/DD, gdzie RR to dwie ostatnie cyfry roku, MM — liczba określająca miesiąc, a DD — dzień miesiąca. W bazie danych tak naprawdę przechowywane są wszystkie cztery cyfry roku, domyślnie jednak są wyświetlane tylko dwie ostatnie.

Choć nazwy kolumn i tabel można wpisywać zarówno małymi, jak i wielkimi literami, lepiej trzymać się jednego sposobu. W przykładach będziemy stosowali wielkie litery dla słów kluczowych SQL i Oracle, a w pozostałych przypadkach — małe litery.

Pobieranie wszystkich kolumn z tabeli

Jeżeli chcemy pobrać wszystkie kolumny z tabeli, możemy użyć gwiazdki (*) zamiast listy kolumn. W powyższym przykładzie gwiazdka została użyta do pobrania wszystkich kolumn z tabeli customers:

```
SELECT *
FROM customers;

CUSTOMER_ID FIRST_NAME LAST_NAME   DOB      PHONE
-----  -----
1 Jan        Nikiel     65/01/01 800-555-1211
2 Lidia      Stal       68/02/05 800-555-1212
3 Stefan     Brąz       71/03/16 800-555-1213
4 Grażyna   Cynk       800-555-1214
5 Jadwiga    Mosiądz   70/05/20
```

Z tabeli customers zostały pobrane wszystkie kolumny.

Wykorzystanie klauzuli WHERE do wskazywania wierszy do pobrania

Klauzula WHERE ogranicza zakres pobieranych wierszy. Oracle może przechowywać w tabeli ogromną liczbę wierszy, a możemy być zainteresowani jedynie niewielkim podzbiorem. Klauzulę WHERE umieszczamy za klauzulą FROM:

```
SELECT lista_elementów
FROM lista_tabel
WHERE lista_warunków;
```

W poniższym zapytaniu użyto klauzuli WHERE do pobrania z tabeli customers wiersza, w którym kolumna customer_id ma wartość 2:

```
SELECT *
FROM customers
WHERE customer_id = 2;

CUSTOMER_ID FIRST_NAME LAST_NAME   DOB      PHONE
-----  -----
2 Lidia      Stal       68/02/05 800-555-1212
```

Identyfikatory wierszy

Każdy wiersz w bazie danych Oracle posiada unikatowy identyfikator nazwany *rowid*, który jest wykorzystywany wewnętrznie przez bazę do przechowywania fizycznej lokalizacji wiersza. Identyfikator wiersza

jest 18-cyfrową liczbą zakodowaną algorytmem base-64. Można przejrzeć identyfikatory wierszy w tabeli, pobierając w zapytaniu kolumnę ROWID.

Na przykład poniższe zapytanie pobiera kolumny ROWID i customer_id z tabeli customers. Należy zwrócić uwagę, że w wynikach otrzymujemy liczby zakodowane base-64:

```
SELECT ROWID, customer_id
FROM customers;
```

ROWID	CUSTOMER_ID
AAADZIAABAAAKViAAA	1
AAADZIAABAAAKViAAB	2
AAADZIAABAAAKViAAC	3
AAADZIAABAAAKViAAD	4
AAADZIAABAAAKViAAE	5

Jeżeli wyświetlamy opis tabeli za pomocą polecenia DESCRIBE w SQL*Plus, kolumna ROWID nie jest pokazywana, ponieważ jest ona wykorzystywana wewnętrznie przez bazę danych. ROWID nazywamy **pseudokolumną**. Poniżej znajduje się opis tabeli customers; należy zwrócić uwagę na brak kolumny ROWID:

```
DESCRIBE customers;
Name          Null?    Type
-----        -----
CUSTOMER_ID   NOT NULL NUMBER(38)
FIRST_NAME    NOT NULL VARCHAR2(10)
LAST_NAME     NOT NULL VARCHAR2(10)
DOB           DATE
PHONE         VARCHAR2(12)
```

Numery wierszy

Kolejną pseudokolumną jest ROWNUM, która zwraca numer wiersza w zestawie wyników. Pierwszy wiersz zwrocony przez zapytanie ma numer 1, drugi wiersz ma numer 2 itd.

Na przykład poniższe zapytanie uwzględnia kolumnę ROWNUM przy pobieraniu wierszy z tabeli customers:

```
SELECT ROWNUM, customer_id, first_name, last_name
FROM customers;
```

ROWNUM	CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	1	Jan	Nikiel
2	2	Lidia	Stał
3	3	Stefan	Brąz
4	4	Grażyna	Cynk
5	5	Jadwiga	Mosiądz

A oto inny przykład:

```
SELECT ROWNUM, customer_id, first_name, last_name
FROM customers
WHERE customer_id = 3;
```

ROWNUM	CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	3	Stefan	Brąz

Wykonywanie działań arytmetycznych

Oracle umożliwia wykonywanie obliczeń arytmetycznych w instrukcjach SQL za pomocą wyrażeń arytmetycznych. Dopuszczalne jest dodawanie, odejmowanie, mnożenie i dzielenie. Wyrażenia arytmetyczne składają się z dwóch **operandów**: liczb lub dat, oraz **operatora** arytmetycznego. W tabeli 2.1 przedstawiono cztery operatory arytmetyczne.

Tabela 2.1. Operatory arytmetyczne

Operator	Opis
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie

Poniższe zapytanie pokazuje, jak za pomocą operatora mnożenia (*) obliczyć iloczyn 2 i 6 (liczby 2 i 6 są operandami):

```
SELECT 2*6
FROM dual;
```

```
2*6
-----
12
```

Został zwrocony prawidłowy wynik. Użycie 2*6 w tym zapytaniu jest przykładem **wyrażenia**. Może ono zawierać kombinację kolumn, wartości literałowych i operatorów.

Wykonywanie obliczeń na danych

Operatory dodawania i odejmowania mogą być używane z datami. Do daty można dodać liczbę reprezentującą liczbę dni. W poniższym przykładzie dodano dwa dni do 25 lipca 2007:

```
SELECT TO_DATE('2007-lip-25') + 2
FROM dual;
```

```
TO_DATE(
-----
07/07/27
```

 **Uwaga** TO_DATE() jest funkcją konwertującą zapis na datę. Zostanie ona dokładniej opisana w rozdziale 5.

Tabela dual

Tabela dual jest często używana w połączeniu z funkcjami i wyrażeniami zwracającymi wartości bez konieczności odwoływanego się do innej tabeli w zapytaniu. Może być na przykład użyta w zapytaniu zwracającym wynik wyrażenia arytmetycznego lub wywołującym funkcję taką jak TO_DATE(). Oczywiście jeśli wyrażenie lub funkcja odwołuje się do kolumny w innej tabeli, nie powinno się w takim zapytaniu używać tabeli dual.

Poniżej przedstawiono wynik polecenia DESCRIBE, przedstawiającego strukturę tabeli dual, która zawiera jedną kolumnę typu VARCHAR2 nazwaną dummy:

```
DESCRIBE dual;
Name          Null?    Type
-----        -----
DUMMY          VARCHAR2(1)
```

Poniższe zapytanie pobiera wiersz z tabeli dual i pokazuje, że w kolumnie dummy znajduje się jeden wiersz zawierający literę X:

```
SELECT *
FROM dual;
```

```
D
-
X
```

Kolejne zapytanie odejmuje trzy dni od 2 sierpnia 2007:

```
SELECT TO_DATE('2007-sie-02') - 3
FROM dual;
```

```
TO_DATE(
-----
07/07/30
```

Można również odejmować daty od siebie, uzyskując w ten sposób liczbę dni między nimi. Poniżej „odejęto” 25 lipca 2007 od 2 sierpnia 2007:

```
SQL> SELECT TO_DATE('2007-sie-02') - TO_DATE('2007-lip-25')
FROM dual;
```

```
TO_DATE('2007-SIE-02')-TO_DATE('2007-LIP-25')
-----
8
```

Korzystanie z kolumn w obliczeniach

Operandami nie muszą być literały liczb czy dat. Mogą to być również kolumny z tabeli. W kolejnym zapytaniu z tabeli products są pobierane kolumny name i price. Należy zauważyć, że do wartości z kolumny price jest dodawane 2 — za pomocą operatora dodawania (+) utworzono wyrażenie price + 2:

```
SELECT name, price + 2
FROM products;
```

NAME	PRICE+2
Nauka współczesna	21,95
Chemia	32
Supernowa	27,99
Wojny czołgów	15,95
Z Files	51,99
2412: Powrót	16,95
Space Force 9	15,49
Z innej planety	14,99
Muzyka klasyczna	12,99
Pop 3	17,99
Twórczy wrzask	16,99
Pierwsza linia	15,49

W jednym wyrażeniu można również umieścić kilka operatorów. W poniższym zapytaniu wartość z kolumny price jest mnożona razy 3, a następnie do wyniku jest dodawane 1:

```
SELECT name, price * 3 + 1
FROM products;
```

NAME	PRICE*3+1
Nauka współczesna	60,85
Chemia	91
Supernowa	78,97
Wojny czołgów	42,85
Z Files	150,97
2412: Powrót	45,85
Space Force 9	41,47
Z innej planety	39,97
Muzyka klasyczna	33,97
Pop 3	48,97
Twórczy wrzask	45,97
Pierwsza linia	41,47

Kolejność wykonywania działań

W SQL obowiązują zwykłe zasady kolejności wykonywania działań: najpierw jest wykonywane dzielenie i mnożenie, a następnie dodawanie i odejmowanie. Jeżeli zostały użyte operatory o tym samym stopniu pierwszeństwa, są one wykonywane od lewej do prawej.

Na przykład w wyrażeniu $10 * 12 / 3 - 1$ najpierw zostanie wykonane mnożenie 10 razy 12, dając w wyniku 120. Następnie 120 zostanie podzielone przez 3, a na koniec od 40 zostanie odjęte 1 i otrzymamy 39:

```
SELECT 10 * 12 / 3 - 1
FROM dual;
```

```
10*12/3-1
-----
39
```

Do określenia kolejności wykonywania działań można również użyć nawiasów ():

```
SELECT 10 * (12 / 3 - 1)
FROM dual;
```

```
10*(12/3-1)
-----
30
```

W powyższym przykładzie nawiasy wymuszają wcześniejsze wykonanie wyrażenia $12 / 3 - 1$ i dopiero jego wynik jest mnożony razy 10, co daje ostateczny wynik 30.

Używanie aliasów kolumn

Gdy pobieramy kolumnę z tabeli, w wynikach jako nagłówek kolumny Oracle wpisuje nazwę kolumny wielkimi literami. Na przykład gdy pobieramy kolumnę price, nagłówkiem w wynikach jest PRICE. Jeżeli wykorzystujemy wyrażenie, Oracle usuwa z niego spację i stosuje je jako nagłówek.

Można określić własny nagłówek, stosując **alias**. W poniższym zapytaniu wyrażenie `price * 2` otrzymuje alias DOUBLE_PRICE:

```
SELECT price * 2 DOUBLE_PRICE
FROM products;
```

```
DOUBLE_PRICE
-----
39,9
60
51,98
27,9
99,98
29,9
26,98
25,98
21,98
31,98
29,98
26,98
```

Jeżeli chcemy użyć spacji i zachować wielkość liter w aliasie, należy umieścić jego tekst w cudzysłowie:

```
SELECT price * 2 "Double Price"
FROM products;
```

```
Double Price
-----
39,9
...
```

Przed tekstem aliasu można również umieścić opcjonalne słowo kluczowe AS:

```
SELECT 10 * (12 / 3 - 1) AS "Obliczenia"
FROM dual;
```

```
Obliczenia
-----
30
```

Łączenie wartości z kolumn za pomocą konkatenacji

Za pomocą konkatenacji można łączyć wartości z kolumn pobrane przez zapytanie, co umożliwia uzyskanie bardziej sensownych wyników. Na przykład w tabeli `customers` kolumny `first_name` i `last_name` zawierają imię i nazwisko (nazwę) klienta. Możemy połączyć te dwie nazwy za pomocą operatora konkatenacji (`||`), co zaprezentowano w poniższym zapytaniu. Należy zwrócić uwagę, że po kolumnie `first_name` został dodany znak spacji, a dopiero później kolumna `last_name`:

```
SELECT first_name || ' ' || last_name AS "Nazwa klienta"
FROM customers;
```

```
Nazwa klienta
-----
Jan Nikiel
Lidia Stal
Stefan Brąz
Grażyna Cynk
Jadwiga Mosiądz
```

Wartości kolumn `first_name` i `last_name` zostały połączone w wynikach pod aliasem `Nazwa klienta`.

Wartości null

W jaki sposób w bazie danych jest reprezentowana nieznana wartość? Jest w tym celu wykorzystywana wartość specjalna — `null`. Nie jest ona pustym wpisem — oznacza, że wartość dla danej kolumny jest nieznana.

Jeżeli pobieramy kolumnę, która zawiera wartość `null`, w jej wynikach nie zobaczymy nic.

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Klient nr 4 ma wartość `null` w kolumnie `dob`, a klient nr 5 w kolumnie `phone`.

Można również sprawdzić występowanie wartości `null`, umieszczając w zapytaniu `IS NULL`. W poniższym przykładzie zostały zwrócone informacje o kliencie nr 4, ponieważ w tym wierszu w kolumnie `dob` występuje wartość `null`:

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE dob IS NULL;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
4	Grażyna	Cynk	

W kolejnym przykładzie zostały zwrócone informacje o kliencie nr 5, ponieważ w jego przypadku phone ma wartość null:

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE phone IS NULL;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
5	Jadwiga	Mosiądz	70/05/20

Ponieważ wartości null nie są w żaden sposób wyświetlane, jak odróżnić je od pustych wpisów? Możemy posłużyć się wbudowaną funkcją Oracle NVL(). Zwraca ona inną wartość w miejscu null. Funkcja NVL() przyjmuje dwa parametry: kolumnę (czy też bardziej ogólnie: dowolne wyrażenie, którego wynikiem jest wartość) oraz wartość, która będzie zwracana, jeżeli pierwszy parametr okaże się być wartością null. W poniższym zapytaniu NVL() zwraca wyrażenie 'Nieznany numer telefonu', jeżeli kolumna phone zawiera wartość null:

```
SELECT customer_id, first_name, last_name,
NVL(phone, 'Nieznany numer telefonu') AS PHONE_NUMBER
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER
1	Jan	Nikiel	800-555-1211
2	Lidia	Stal	800-555-1212
3	Stefan	Brąz	800-555-1213
4	Grażyna	Cynk	800-555-1214
5	Jadwiga	Mosiądz	Nieznany numer telefonu

Funkcję NVL() można również zastosować do konwersji pustych danych liczbowych i dat. W poniższym zapytaniu NVL() zwraca datę 1 stycznia 2000, jeżeli kolumna dob zawiera wartość null:

```
SELECT customer_id, first_name, last_name,
NVL(dob, '2000-sty-01') AS DOB
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	Jan	Nikiel	65/01/01
2	Lidia	Stal	68/02/05
3	Stefan	Brąz	71/03/16
4	Grażyna	Cynk	00/01/01
5	Jadwiga	Mosiądz	70/05/20

Należy zwrócić uwagę, że w kolumnie dob dla klienta nr 4 jest wyświetlana wartość 00/01/01.

Wyświetlanie unikatowych wierszy

Załóżmy, że chcemy uzyskać listę klientów, którzy dokonali zakupów w naszym sklepie. Możemy to zrobić za pomocą poniższego zapytania, które pobiera kolumnę customer_id z tabeli purchases:

```
SELECT customer_id
FROM purchases;
```

CUSTOMER_ID
1
2
3
4
1
2

3
4
3

Kolumna `customer_id` zawiera identyfikatory klientów, którzy dokonali zakupu w sklepie. Jak widzimy w wynikach zwróconych przez zapytanie, niektórzy klienci kupili kilka rzeczy i w związku z tym ich identyfikator się powtarza.

Można usunąć powtarzające się wiersze za pomocą słowa kluczowego `DISTINCT`. W poniższym zapytaniu użyto go, aby pominąć powtarzające się wiersze:

```
SELECT DISTINCT customer_id
FROM purchases;
```

CUSTOMER_ID
1
2
4
3

Na tej liście łatwiej zauważyc, że zakupów dokonali klienci nr 1, 2, 3 i 4; powtarzające się wiersze zostały pominięte.

Porównywanie wartości

Tabela 2.2 zawiera listę operatorów używanych do porównywania wartości.

Tabela 2.2. Operatory służące do porównywania wartości

Operator	Opis
=	Równe
<> lub !=	Nierówne Powinno się używać <code><></code> , ponieważ jest to zgodne ze standardem ANSI
<	Mniejsze niż
>	Większe niż
<=	Mniejsze lub równe
>=	Większe lub równe
ANY	Porównuje jedną wartość z dowolnymi wartościami z listy
SOME	Takie samo znaczenie jak ANY. Należy stosować operator ANY, ponieważ jest on częściej wykorzystywany oraz bardziej zrozumiały
ALL	Porównuje jedną wartość ze wszystkimi wartościami z listy

Operator <>

W klauzuli `WHERE` poniższego zapytania użyto operatora nierówności (`<>`), aby pobrać z tabeli `customers` wiersze, których `customer_id` jest różny od 2:

```
SELECT *
FROM customers
WHERE customer_id <> 2;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Operator >

W kolejnym zapytaniu użyto operatora > do pobrania kolumn product_id i name z tabeli products, gdzie wartość w kolumnie product_id jest większa od 8:

```
SELECT product_id, name
FROM products
WHERE product_id > 8;
```

PRODUCT_ID	NAME
9	Muzyka klasyczna
10	Pop 3
11	Twórczy wrzask
12	Piewsza linia

Operator <=

W poniższym zapytaniu wykorzystano pseudokolumnę ROWNUM i operator <=, aby pobrać pierwsze trzy wiersze z tabeli products:

```
SELECT ROWNUM, product_id, name
FROM products
WHERE ROWNUM <= 3;
```

ROWNUM	PRODUCT_ID	NAME
1	1	Nauka współczesna
2	2	Chemia
3	3	Supernowa

Operator ANY

W celu porównania wartości z *którkolwiek* z listy należy użyć operatora ANY w klauzuli WHERE. Przed słowem kluczowym ANY należy umieścić operator =, <, >, <= lub >=. W poniższym zapytaniu użyto operatora ANY w celu pobrania z tabeli customers wierszy, w których wartość w kolumnie customer_id jest większa niż którakolwiek z liczb 2, 3 lub 4:

```
SELECT *
FROM customers
WHERE customer_id > ANY (2, 3, 4);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Operator ALL

W celu porównania wartości ze *wszystkimi* wartościami z listy należy użyć operatora ALL. Przed słowem kluczowym ALL należy umieścić operator =, <, >, <= lub >=. W poniższym zapytaniu użyto operatora ALL w celu pobrania z tabeli customers wierszy, w których wartość w kolumnie customer_id jest większa niż każda z liczb 2, 3 lub 4:

```
SELECT *
FROM customers
WHERE customer_id > ALL (2, 3, 4);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
5	Jadwiga	Mosiądz	70/05/20	

Został zwrócony jedynie wiersz dla klienta nr 5, ponieważ 5 jest większe od 2, 3 i 4.

Korzystanie z operatorów SQL

Operatory SQL umożliwiają ograniczenie liczby zwróconych wierszy na podstawie dopasowań do wzorca wpisów, list wartości, zakresów wartości i wartości null. Operatory SQL zostały wymienione w tabeli 2.3.

Tabela 2.3. Operatory SQL

Operator	Opis
LIKE	Porównuje wpisy ze wzorem
IN	Porównuje wartości z listą
BETWEEN	Porównuje wartości z zakresem
IS NULL	Sprawdza, czy wartość to null
IS NAN	Spełniany przez wartość specjalną NAN, czyli „nieliczbę”
IS INFINITE	Spełniany przez nieskończone wartości BINARY_FLOAT i BINARY_DOUBLE

Za pomocą słowa kluczowego NOT można odwrócić znaczenie operatora:

- NOT LIKE,
- NOT IN,
- NOT BETWEEN,
- IS NOT NULL,
- IS NOT NAN,
- IS NOT INFINITE.

Kolejne podrozdziały zawierają informacje o operatorach LIKE, IN i BETWEEN.

Operator LIKE

Operator LIKE w klauzuli WHERE służy do wyszukiwania wzorca wpisu. Wzorce są definiowane za pomocą kombinacji zwykłych znaków oraz poniższych dwóch symboli wieloznacznych:

- znaku podkreślenia (_), który oznacza jeden znak w określonym miejscu,
- procentu (%) oznaczającego dowolną liczbę znaków, począwszy od określonego miejsca.

Rozważmy następujący wzorzec:

'_a%'

Znak podkreślenia (_) jest spełniany przez dowolny znak na pierwszej pozycji, a jest spełniane przez znak „a” na drugiej pozycji, a znak procentu (%) — przez dowolne znaki znajdujące się za znakiem a. W poniższym zapytaniu użyto operatora LIKE do wyszukania wzorca '_a%' w kolumnie first_name tabeli customers:

```
SELECT *
FROM customers
WHERE first_name LIKE '_a%';

CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
----- -----
1 Jan Nikiel 65/01/01 800-555-1211
5 Jadwiga Mosiądz 70/05/20
```

Zostały zwrócone dwa wiersze, ponieważ tylko we wpisach Jan i Jadwiga znak a stoi na drugiej pozycji.

W kolejnym zapytaniu użyto NOT LIKE w celu pobrania wierszy, które nie zostały zwrócone przez poprzednie zapytanie:

```
SELECT *
FROM customers
WHERE first_name NOT LIKE '_a%';
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
-----+
 2 Lidia      Stal     68/02/05 800-555-1212
 3 Stefan     Brąz    71/03/16 800-555-1213
 4 Grażyna   Cynk      800-555-1214
```

Jeżeli chcemy wyszukać we wpisie znak podkreślenia lub procentu, możemy użyć opcji `ESCAPE` do określenia tych znaków. Rozważmy poniższy wzorzec:

```
'%\%%' ESCAPE '\'
```

Znak za opcją `ESCAPE` pozwala odróżnić znaki do wyszukania od symboli wieloznacznych — w przykładzie został użyty lewy ukośnik (\). Pierwszy znak % jest traktowany jako symbol wieloznaczny, spełniały przez dowolną liczbę znaków, drugi znak % — jako znak do wyszukania, a trzeci jest traktowany jako symbol wieloznaczny, spełniany przez dowolną liczbę znaków.

W poniższym zapytaniu jest wykorzystywana tabela `promotions`, zawierająca szczegółowe dane o produktach wycofywanych ze sprzedaży (więcej informacji na temat tej tabeli znajduje się w dalszej części książki). Zapytanie wykorzystuje operator `LIKE` do wyszukania w kolumnie `name` tabeli `promotions` wzorca `'%\%%'` `ESCAPE '\'`:

```
SELECT name
FROM promotions
WHERE name LIKE '%\%%' ESCAPE '\';
```

```
NAME
-----
10% taniej Z Files
20% taniej Pop 3
30% taniej Nauka współczesna
20% taniej Wojny czołgów
10% taniej Chemia
20% taniej Twórczy wrzask
15% taniej Pierwsza linia
```

Zapytanie zwróciło wiersze, których nazwy zawierają znak procentu.

Operator IN

Operator `IN` pozwala wybrać wartości znajdujące się na liście. W poniższym zapytaniu użyto operatora `IN` do pobrania z tabeli `customers` wierszy, dla których `customer_id` ma wartość 2, 3 lub 5:

```
SELECT *
FROM customers
WHERE customer_id IN (2, 3, 5);
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
-----+
 2 Lidia      Stal     68/02/05 800-555-1212
 3 Stefan     Brąz    71/03/16 800-555-1213
 5 Jadwiga   Mosiądz  70/05/20
```

`NOT IN` pobierze wiersze, które nie zostały pobrane przy użyciu operatora `IN`:

```
SELECT *
FROM customers
WHERE customer_id NOT IN (2, 3, 5);
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
-----+
 1 Jan        Nikiel   65/01/01 800-555-1211
 4 Grażyna   Cynk      800-555-1214
```

Należy pamiętać, że `NOT IN` zwraca fałsz, jeżeli na liście znajduje się wartość `NULL`. Obrazuje to poniższe zapytanie, które nie zwraca żadnych wierszy, ponieważ na liście znajduje się ta wartość:

```
SELECT *
FROM customers
WHERE customer_id NOT IN (2, 3, 5, NULL);
```

nie wybrano żadnych wierszy

 **Ostrzeżenie** NOT IN zwraca fałsz, jeżeli na liście znajduje się wartość NULL. Jest to ważne, ponieważ możemy na niej umieścić dowolne wyrażenie, a nie tylko litery, dostrzeżenie miejsca, w którym wystąpiła wartość NULL, może więc być trudne. W przypadku wyrażeń, które mogą zwrócić wartość NULL, warto zastanowić się nad zastosowaniem funkcji NVL().

Operator BETWEEN

Operator BETWEEN w klauzuli WHERE pozwala wybrać wiersze, w których wartość z danej kolumny zawiera się w określonym przedziale. Jest on domknięty, co oznacza, że obydwa końce należą do przedziału. W późniejszym zapytaniu użyto operatora BETWEEN do pobrania z tabeli customers wierszy, w których customer_id należy do przedziału od 1 do 3:

```
SELECT *
FROM customers
WHERE customer_id BETWEEN 1 AND 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213

NOT BETWEEN pobiera wiersze, które nie zostały pobrane przy użyciu operatora BETWEEN:

```
SELECT *
FROM customers
WHERE customer_id NOT BETWEEN 1 AND 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Operatory logiczne

Operatory logiczne pozwalają ograniczyć liczbę zwracanych wierszy za pomocą warunków logicznych. Operatory logiczne zostały opisane w tabeli 2.4.

Tabela 2.4. Operatory logiczne

Operator	Opis
$x \text{ AND } y$	Zwraca prawdę, jeżeli x i y są jednocześnie prawdziwe
$x \text{ OR } y$	Zwraca prawdę, jeżeli prawdziwe jest x lub y
NOT x	Zwraca prawdę, jeżeli x nie jest prawdą i zwraca fałsz, jeżeli x jest prawdą

Operator AND

Poniższe zapytanie obrazuje użycie operatora AND do pobrania z tabeli customers wierszy, dla których oba poniższe warunki są prawdziwe:

- wartość w kolumnie dob jest większa niż 1 stycznia 1970,
- wartość w kolumnie customer_id jest większa od 3.

```
SELECT *
FROM customers
WHERE dob > '1970-01-01'
AND customer_id > 3;

CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
----- -----
5 Jadwiga Mosiądz 70/05/20
```

Operator OR

Kolejne zapytanie obrazuje użycie operatora OR do pobrania z tabeli *customers* wierszy, dla których *przynajmniej jeden* z poniższych warunków jest prawdziwy:

- wartość w kolumnie *dob* jest większa niż 1 stycznia 1970,
- wartość w kolumnie *customer_id* jest większa od 3.

```
SELECT *
FROM customers
WHERE dob > '1970-01-01'
OR customer_id > 3;

CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
----- -----
3 Stefan Brąz 71/03/16 800-555-1213
4 Grażyna Cynk 800-555-1214
5 Jadwiga Mosiądz 70/05/20
```

Jak przekonamy się w kolejnym podrozdziale, operatory AND i OR można wykorzystać do łączenia wyrażeń w klauzuli WHERE.

Następstwo operatorów

Jeżeli w tym samym wyrażeniu połączymy operatory AND i OR, operator AND będzie miał pierwszeństwo przed operatorem OR (oznacza to, że jego rezultat zostanie wyliczony jako pierwszy). Operatory porównania mają pierwszeństwo przed operatorem AND. Można zmienić domyślne następstwo operatorów, używając nawiasów do określenia kolejności wykonywania wyrażeń.

W poniższym przykładzie z tabeli *customers* są pobierane wiersze, dla których jest spełniony *którykolwiek* z poniższych warunków:

- wartość w kolumnie *dob* jest większa niż 1 stycznia 1970,
- wartość w kolumnie *customer_id* jest mniejsza od 2 i na końcu wartości w kolumnie *phone* znajduje się 1211.

```
SELECT *
FROM customers
WHERE dob > '1970-sty-01'
OR customer_id < 2
AND phone LIKE '%1211';

CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
----- -----
1 Jan Nikiel 65/01/01 800-555-1211
3 Stefan Brąz 71/03/16 800-555-1213
5 Jadwiga Mosiądz 70/05/20
```

Jak zostało to wspomniane wcześniej, operator AND ma pierwszeństwo przed operatorem OR, więc klauzulę WHERE w powyższym zapytaniu możemy sobie wyobrazić jako:

```
dob > '1970-sty-01' OR (customer_id < 2 AND phone LIKE '%1211')
```

Dlatego też zapytanie zwróciło wiersze klientów nr 1, 3 i 5.

Sortowanie wierszy za pomocą klauzuli ORDER BY

Klauzula ORDER BY służy do sortowania wierszy zwracanych przez zapytanie. Za jej pomocą można określić kilka kolumn, według których będzie się odbywało sortowanie; ponadto klauzula ta musi zostać umieszczona za klauzulami FROM i WHERE (jeżeli ta ostatnia została użyta).

W poniższym zapytaniu użyto klauzuli ORDER BY do posortowania wierszy z tabeli customers według kolumny last_name:

```
SELECT *
FROM customers
ORDER BY last_name;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212

Domyślnie klauzula ORDER BY powoduje posortowanie w kolejności rosnącej (mniejsze wartości wyświetlane są u góry). Aby posortować kolumny w kolejności malejącej (od wartości największych do najmniejszych), należy zastosować słowo kluczowe DESC. Można również użyć słowa kluczowego ASC, aby jawnie zadeklarować rosnącą kolejność sortowania i zwiększyć czytelność zapytania.

W poniższym zapytaniu użyto klauzuli ORDER BY do posortowania wierszy pobranych z tabeli customers — rosnąco według kolumny first_name i malejąco według last_name:

```
SELECT *
FROM customers
ORDER BY first_name ASC, last_name DESC;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213

Do wskazania w klauzuli ORDER BY kolumny, według której będą sortowane wiersze, można również użyć numeru pozycji kolumny — 1 oznacza pierwszą wybieraną kolumnę, 2 — drugą itd. W poniższym zapytaniu kryterium sortowania są dane z kolumny 1 (customer_id):

```
SELECT customer_id, first_name, last_name
FROM customers
ORDER BY 1;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	Jan	Nikiel
2	Lidia	Stal
3	Stefan	Brąz
4	Grażyna	Cynk
5	Jadwiga	Mosiądz

Ponieważ kolumna customer_id znajduje się na pierwszej pozycji za słowem kluczowym SELECT, to ona posłużyła za kryterium sortowania.

Instrukcje SELECT wykorzystujące dwie tabele

Schematy baz danych zawierają zwykle więcej niż jedną tabelę. Na przykład w schemacie store znajdują się tabele przechowujące informacje o klientach, towarach, pracownikach itd. Jak dotąd wszystkie zapytania przedstawione w tej książce pobierały wiersze tylko z jednej tabeli. W rzeczywistości często chcemy pobrać dane z kilku tabel. Możemy na przykład chcieć uzyskać nazwę towaru oraz nazwę kategorii, do której został on przypisany. W tym podrozdziale nauczysz się tworzyć zapytania wykorzystujące dwie tabele. Dowiesz się także, jak wykorzystywać zapytania pracujące na jeszcze większej liczbie tabel.

Powróćmy do przykładu, w którym chcieliśmy pobrać nazwę produktu nr 3 oraz jego kategorię. Nazwa towaru jest przechowywana w kolumnie name tabeli products, a nazwa kategorii — w kolumnie name tabeli product_types. Tabele te są z sobą powiązane za pośrednictwem kolumny klucza obcego product_type_id. Kolumna ta (klucz obcy) w tabeli products wskazuje na kolumnę product_type_id (klucz główny) tabeli product_types.

Poniższe zapytanie pobiera z tabeli products kolumny name i product_type_id dla produktu nr 3:

```
SELECT name, product_type_id
FROM products
WHERE product_id = 3;
```

NAME	PRODUCT_TYPE_ID
Supernowa	2

Następne zapytanie pobiera z tabeli product_types kolumnę name dla product_type_id równego 2:

```
SELECT name
FROM product_types
WHERE product_type_id = 2;
```

NAME
VHS

Dowiedzieliśmy się, że produkt nr 3 to kaseta wideo. Musielimy w tym celu wykonać dwa zapytania.

Można jednak pobrać nazwę produktu i jego kategorii, stosując jedno zapytanie. W takiej sytuacji należy zastosować w zapytaniu **złączenie tabel**. Aby to zrobić, należy dołączyć obie tabele do klauzuli FROM zapytania, a także uwzględnić odpowiednie kolumny ze wszystkich tabel w klauzuli WHERE.

W naszym przykładzie klauzula FROM będzie miała postać:

```
FROM products, product_types
```

Natomiast klauzula WHERE:

```
WHERE products.product_type_id = product_types.product_type_id
AND products.product_id = 3;
```

Złączenie jest pierwszym warunkiem w klauzuli WHERE (products.product_type_id = product_types.product_type_id). Najczęściej w złączeniu są stosowane kolumny będące kluczem głównym jednej tabeli i kluczem obcym drugiej tabeli. Drugi warunek w klauzuli WHERE (products.product_id = 3) pobiera produkt nr 3.

Jak możemy zauważyc, w klauzuli WHERE są umieszczone zarówno nazwy kolumn, jak i tabel. Jest to spowodowane tym, że kolumna product_type_id znajduje się i w tabeli products, i product_types, musimy więc w jakiś sposób określić tabelę z kolumną, której chcemy użyć. Gdyby kolumny miały różne nazwy, moglibyśmy pominąć nazwy tabel, należy jednak zawsze je umieszczać, aby było jasne, skąd pochodzi dana kolumna.

Klauzula SELECT w naszym zapytaniu będzie miała postać:

```
SELECT products.name, product_types.name
```

Zapytanie ma zatem postać:

```
SELECT products.name, product_types.name
FROM products, product_types
```

```
WHERE products.product_type_id = product_types.product_type_id
AND products.product_id = 3;
```

NAME	NAME
Supernowa	VHS

Świetnie! To jedno zapytanie zwraca nazwę produktu i nazwę kategorii.

Kolejne zapytanie pobiera wszystkie produkty i porządkuje je według kolumny `products.name`:

```
SELECT products.name, product_types.name
FROM products, product_types
WHERE products.product_type_id = product_types.product_type_id
ORDER BY products.name;
```

NAME	NAME
2412: Powrót	VHS
Chemia	Książka
Muzyka klasyczna	CD
Nauka współczesna	Książka
Pop 3	CD
Space Force 9	DVD
Supernowa	VHS
Twórczy wrzask	CD
Wojny czołgów	VHS
Z Files	VHS
Z innej planety	DVD

Należy zauważyc, że w wynikach brakuje produktu „Pierwsza linia”. Wartość `product_type_id` w wierszu tego produktu wynosi `null`, a warunek złączenia nie zwraca tego wiersza. Z podrozdziału „Złączenia zewnętrzne” dowiemy się, jak dołączyć ten wiersz.

Dotychczas prezentowana składnia złączeń wykorzystuje składnię Oracle, opartą na standardzie ANSI (*American National Standards Institute*) SQL/86. Od wersji 9g Oracle Database obsługuje również standard składni ANSI SQL/92, który zostanie opisany w podrozdziale „Wykonywanie złączeń za pomocą składni SQL/92”. Podczas pracy z Oracle Database 9g lub nowszą należy używać składni SQL/92, składnię SQL/86 należy natomiast stosować wyłącznie podczas pracy z Oracle Database 8g i wcześniejszymi.

Używanie aliasów tabel

W poprzednim podrozdziale utworzyliśmy następujące zapytanie:

```
SELECT products.name, product_types.name
FROM products, product_types
WHERE products.product_type_id = product_types.product_type_id
ORDER BY products.name;
```

Możemy zauważyc, że nazwy tabel `products` i `product_types` zostały użyte zarówno w klauzuli `SELECT`, jak i `WHERE`. Możliwe jest zdefiniowanie aliasów tabel w klauzuli `FROM` i korzystanie z nich, gdy odwołujemy się do tabel w innych miejscach w zapytaniu.

Na przykład w poniższym zapytaniu użyto aliasu `p` dla tabeli `products` i `pt` dla tabeli `product_types`. Należy zauważyc, że aliasy tabel są definiowane w klauzuli `FROM` i umieszczane przed nazwami kolumn w innych fragmentach zapytania:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
ORDER BY pt.name;
```

Aliasy tabel zwiększą czytelność zapytań, zwłaszcza gdy piszemy długie zapytania, wykorzystujące wiele tabel.

Iloczyny kartezańskie

Jeżeli warunek złączenia nie zostanie zdefiniowany, złączone zostaną wszystkie wiersze z jednej tabeli ze wszystkimi wierszami drugiej. Taki zestaw wyników nazywamy **iloczynem kartezańskim**.

Załóżmy, że w jednej tabeli znajduje się 50 wierszy, a w drugiej 100. Jeżeli wybierzymy kolumny z tych tabel bez warunku złączenia, otrzymamy w wyniku 5 000 wierszy, ponieważ każdy wiersz z pierwszej tabeli zostanie złączony z każdym wierszem z drugiej tabeli. To oznacza, że otrzymamy 50×100 wierszy, czyli 5 000 wierszy.

Poniższy przykład przedstawia fragment iloczynu kartezańskiego tabel product_types i products:

```
SQL> SELECT pt.product_type_id, p.product_id
  FROM product_types pt, products p;
```

PRODUCT_TYPE_ID	PRODUCT_ID
1	1
1	2
1	3
1	4
1	5
...	
5	8
5	9
5	10
5	11
5	12

60 wierszy zostało wybranych.

Zapytanie zwróciło 60 wierszy, ponieważ tabele product_types i products zawierają odpowiednio 5 i 12 wierszy, a $5 \times 12 = 60$.

W niektórych przypadkach iloczyny kartezańskie mogą okazać się przydatne, ale w większości przypadków nie będziesz ich potrzebował, dlatego do zapytania należy dodawać warunki złączenia w miarę potrzeb.

Instrukcje SELECT wykorzystujące więcej niż dwie tabele

Złączenia mogą obejmować dowolną liczbę tabel. Liczbę złączeń potrzebnych w klauzuli WHERE możemy obliczyć według poniższego wzoru:

$$\text{Liczba złączeń} = \text{liczba tabel wykorzystywanych w zapytaniu} - 1$$

Na przykład poniższe zapytanie wykorzystuje dwie tabele i dlatego użyte jest jedno złączenie:

```
SELECT p.name, pt.name
  FROM products p, product_types pt
 WHERE p.product_type_id = pt.product_type_id
 ORDER BY p.name;
```

Rozważmy bardziej skomplikowany przykład, wykorzystujący cztery tabele:

- dane o zakupach dokonanych przez klientów (z tabeli purchases),
- imię i nazwisko klienta (z tabeli customers),
- nazwę zakupionego towaru (z tabeli products),
- nazwę kategorii towaru (z tabeli product_types).

Korzystamy z czterech tabel i dlatego potrzebujemy trzech złączeń. Poniżej zostały wymienione kolejne złączenia:

- Aby uzyskać dane klienta, który dokonał zakupu, musimy złączyć tabele `customers` i `purchases`, wykorzystując kolumny `customer_id` (`customers.customer_id = purchases.customer_id`).
- Aby dowiedzieć się, jaki produkt został zakupiony, musimy złączyć tabele `products` i `purchases`, używając kolumny `product_id` (`products.product_id = purchases.product_id`).
- Aby uzyskać nazwę kategorii produktu, musimy złączyć tabele `products` i `product_types`, wykorzystując kolumny `product_type_id` (`products.product_type_id = product_types.product_type_id`).

Te złączenia zostały zastosowane w poniższym zapytaniu:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c, purchases pr, products p, product_types pt
WHERE c.customer_id = pr.customer_id
AND p.product_id = pr.product_id
AND p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

FIRST_NAME	LAST_NAME	PRODUCT	TYPE
Grażyna	Cynk	Chemia	Książka
Lidia	Stał	Chemia	Książka
Jan	Nikiel	Chemia	Książka
Stefan	Brąz	Chemia	Książka
Lidia	Stał	Nauka współczesna	Książka
Stefan	Brąz	Nauka współczesna	Książka
Jan	Nikiel	Nauka współczesna	Książka
Grażyna	Cynk	Nauka współczesna	Książka
Stefan	Brąz	Supernowa	VHS

Prezentowane dotychczas zapytania pobierające dane z wielu tabel wykorzystywały w warunkach złączenia operator równości (=), były to więc **równozłączenia**. Jak przekonamy się w kolejnym podrozdziale, nie jest to jedyny typ złączeń.

Warunki złączenia i typy złączeń

W tym podrozdziale poznamy warunki i typy złączeń, umożliwiające tworzenie bardziej zaawansowanych zapytań.

Istnieją dwa rodzaje warunków złączeń, wyróżniane na podstawie użytego operatora:

- W **równozłączeniach** jest wykorzystywany operator równości (=).
- W **nierównozłączeniach** jest wykorzystywany inny operator niż operator równości, na przykład <, >, BETWEEN itd.

Występują również trzy typy złączeń:

- **Złączenia wewnętrzne** zwracają wiersz *tylko wtedy*, gdy kolumny w złączeniu spełniają warunek złączenia. To oznacza, że jeżeli wiersz w jednej z kolumn w warunku złączenia posiada wartość NULL, nie zostanie on zwrócony. Prezentowane dotychczas złączenia były przykładami złączeń wewnętrznych.
- **Złączenia zewnętrzne** zwracają wiersz, *nawet jeżeli* jedna z kolumn warunku zawiera wartość NULL.
- **Złączenia własne** zwracają wiersze złączone w tej samej tabeli.

Omówię teraz nierównozłączenia, złączenia zewnętrzne i własne.

Nierównozłączenia

W nierównozłączaniu warunek jest określony za pomocą innego operatora niż operator równości (=). Może być to operator nierówności (<>), mniejsze niż (<), większe niż (>), mniejsze lub równe (<=), większe lub równe (>=), LIKE, IN oraz BETWEEN.

Załóżmy, że chcemy uzyskać przedziały wysokości wynagrodzeń pracowników. Poniższe zapytanie pobiera przedziały wynagrodzeń z tabeli salary_grades:

```
SELECT *
FROM salary_grades;
```

SALARY_GRADE_ID	LOW_SALARY	HIGH_SALARY
1	1	250000
2	250001	500000
3	500001	750000
4	750001	999999

W kolejnym zapytaniu wykorzystano nierównozłączenie w celu pobrania wynagrodzeń oraz przedziałów wynagrodzeń pracowników. Przedział wynagrodzenia jest wybierany za pomocą operatora BETWEEN:

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e, salary_grades sg
WHERE e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

FIRST_NAME	LAST_NAME	TITLE	SALARY	SALARY_GRADE_ID
Fryderyk	Helc	Sprzedawca	150000	1
Zofia	Nowak	Sprzedawca	500000	2
Roman	Joświerz	Kierownik sprzedaży	600000	3
Jan	Kowalski	Prezes	800000	4

W tym zapytaniu operator BETWEEN zwraca prawdę, jeżeli wynagrodzenie pracownika mieści się między dolną i górną granicą przedziału wynagrodzeń (w tabeli salary_grades są to kolumny low_salary i high_salary). Jeżeli zostanie zwrócona prawda, wyszukiwany przedział wynagrodzeń jest przedziałem pracownika.

Na przykład wynagrodzenie Fryderyka Helca wynosi 150 000 zł, więc mieści się w przedziale z salary_grade_id równym 1, wyznaczanym przez granice 1 zł i 250 000 zł. Wynagrodzenie Zofii Nowak (500 000 zł) mieści się między dolną (250 001 zł) i górną (500 000 zł) granicą przedziału wynagrodzeń o identyfikatorze 2, dlatego należy do przedziału 2.

Wynagrodzenia Romana Joświerza i Jana Kowalskiego należą odpowiednio do przedziałów 3. i 4.

Złączenia zewnętrzne

Złączenie zewnętrzne (ang. *outer join*) pobiera wiersz, nawet jeżeli jedna z jego kolumn zawiera wartość NULL. W celu utworzenia go należy w warunku złączenia zastosować odpowiedni operator. W Oracle operatorem złączeń złożonych jest znak plus umieszczony w nawiasach (+).

Jak pamiętamy, jedno z wcześniejszych zapytań nie zwracało produktu „Pierwsza linia”, ponieważ product_type_id ma wartość NULL. Do pobrania tego wiersza możemy wykorzystać złączenie zewnętrzne:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id (+)
ORDER BY p.name;
```

NAME	NAME
2412: Powrót	VHS
Chemia	Książka
Muzyka klasyczna	CD
Nauka współczesna	Książka
Pierwsza linia	
Pop 3	CD
Space Force 9	DVD
Supernowa	VHS
Twórczy wrzask	CD
Wojny czołgów	VHS

Z Files	VHS
Z innej planety	DVD

Należy zauważać, że towar „Pierwsza linia” został pobrany, chociaż w jego przypadku `product_type_id` ma wartość `NULL`.

Klauzula `WHERE` ma tutaj postać:

```
WHERE p.product_type_id = pt.product_type_id (+)
```

Operator złączenia zewnętrznego znajduje się po prawej stronie operatora równozłączenia, a kolumna `p.product_type_id` z tabeli `product` zawierająca wartość `NULL` znajduje się po lewej stronie operatora równozłączenia.



Operator złączenia zewnętrznego (`+`) należy umieścić po przeciwej stronie operatora równozłączenia (`=`), niż znajduje się kolumna zawierająca wartość `NULL`.

Poniższe zapytanie zwraca takie same wyniki jak poprzednie, ale operator złączenia zewnętrznego znajduje się po lewej stronie operatora równozłączenia, a kolumna z wartością `NULL` po prawej stronie tego operatora:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE pt.product_type_id (+) = p.product_type_id
ORDER BY p.name;
```

Lewo- i prawostronne złączenia zewnętrzne

Wyróżniamy dwa typy złączeń zewnętrznych:

- lewostronne złączenia zewnętrzne,
- prawostronne złączenia zewnętrzne.

Aby zrozumieć różnicę między lewo- i prawostronnymi złączeniami zewnętrznyimi, rozważmy następującą składnię:

```
SELECT ...
FROM tabela1, tabela2
...
```

Załóżmy, że tabele będą łączone według `tabela1.kolumna1` i `tabela2.kolumna2`, a `tabela1` zawiera wiersz, dla którego `kolumna1` ma wartość `NULL`. Przy lewostronnym złączeniu zewnętrznym klauzula `WHERE` będzie miała postać:

```
WHERE tabela1.kolumna1 = tabela2.kolumna2 (+);
```



W przypadku lewostronnego złączenia zewnętrznego operator złączenia znajduje się po prawej stronie operatora równości.

Teraz założmy, że `tabela2` zawiera wiersz z wartością `NULL` w `kolumna2`. Aby wykonać prawostronne złączenie zewnętrzne, musimy przestawić operator złączenia zewnętrznego na lewą stronę operatora równości. W związku z tym klauzula `WHERE` będzie miała postać:

```
WHERE tabela1.kolumna1 (+) = tabela2.kolumna2
```



Jak się przekonamy, jeżeli zarówno `tabela1`, jak i `tabela2` będą zawierały wiersze z wartościami `NULL`, będziemy otrzymywali różne wyniki w zależności od tego, czy zostanie zastosowane lewo-, czy też prawostronne złączenie zewnętrzne.

Przejdźmy do innych przykładów, aby dokładnie wyjaśnić różnicę między lewo- i prawostronnymi złączeniami zewnętrznyimi.

Przykład lewostronnego złączenia zewnętrznego

W poniższym zapytaniu wykorzystano lewostronne złączenie zewnętrzne. Operator tego złączenia znajduje się po prawej stronie operatora równości:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id (+)
ORDER BY p.name;
```

NAME	NAME
2412: Powrót	VHS
Chemia	Książka
Muzyka klasyczna	CD
Nauka współczesna	Książka
Pierwsza linia	
Pop 3	CD
Space Force 9	DVD
Supernowa	VHS
Twórczy wrzask	CD
Wojny czołgów	VHS
Z Files	VHS
Z innej planety	DVD

Z tabeli products zostały pobrane wszystkie wiersze, łącznie z wierszem „Pierwsza linia”, w którym kolumna p.product_type_id ma wartość NULL.

Przykład prawostronnegołączenia zewnętrznego

Tabela product_types zawiera kategorię produktu, do której nie ma odwołania w tabeli products (nie ma w niej czasopism). Kategoria czasopism pojawia się na końcu poniższego przykładu:

```
SELECT *
FROM product_types;
```

PRODUCT_TYPE_ID	NAME
1	Książka
2	VHS
3	DVD
4	CD
5	Czasopismo

Wykorzystując prawostronnełączenie zewnętrzne, możemy pobrać czasopismo w łączaniu tabel products i product_types, co obrazuje poniższe zapytanie. Operatorłączenia zewnętrzneznajduje się po lewej stronie operatora równości:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id (+) = pt.product_type_id
ORDER BY p.name;
```

NAME	NAME
2412: Powrót	VHS
Chemia	Książka
Muzyka klasyczna	CD
Nauka współczesna	Książka
Pop 3	CD
Space Force 9	DVD
Supernowa	VHS
Twórczy wrzask	CD
Wojny czołgów	VHS
Z Files	VHS
Z innej planety	Czasopismo

Ograniczenia złączeń zewnętrznych

Z używaniem złączeń zewnętrznych wiąże się kilka ograniczeń. Operator złączenia zewnętrznego może znajdować się tylko po jednej stronie operatora równości (nie po obu stronach). Jeżeli spróbujemy umieścić operator złączenia zewnętrznego po obu stronach operatora równości, otrzymamy poniższy komunikat o błędzie:

```
SQL> SELECT p.name, pt.name
  2  FROM products p, product_types pt
  3 WHERE p.product_type_id (+) = pt.product_type_id (+);
WHERE p.product_type_id (+) = pt.product_type_id (+)
      *
```

BŁĄD w linii 3:

ORA-01468: predykat może odwoływać się tylko do jednej łączonej zewnętrznie tabeli

Nie można użyć warunku złączenia zewnętrznego z innym złączeniem, wykorzystując operator OR:

```
SQL> SELECT p.name, pt.name
  2  FROM products p, product_types pt
  3 WHERE p.product_type_id (+) = pt.product_type_id
  4 OR p.product_type_id = 1;
WHERE p.product_type_id (+) = pt.product_type_id
      *
```

BŁĄD w linii 3:

ORA-01719: operator (+) złączenia zewnętrznego nie jest dozwolony w operandzie OR lub IN

 Powyżej zostały opisane najczęściej spotykane ograniczenia przy stosowaniu operatora złączenia zewnętrznego. Wszystkie znajdują się w podręczniku *Oracle Database SQL Reference* firmy Oracle Corporation.

Złączenia własne

Złączenie własne jest dokonywane na tej samej tabeli. W celu wykonania złączenia własnego musimy użyć innego aliasu tabeli, aby zidentyfikować w zapytaniu każde odwołanie do niej. Rozważmy przykład: tabela employees zawiera kolumnę manager_id, która zawiera wartość employee_id kierownika każdego pracownika. Jeżeli pracownik nie ma kierownika, manager_id ma wartość NULL.

Tabela employees zawiera następujące wiersze:

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1	Jan	Kowalski	Prezes	800000	
2	1	Roman	Joświerz	Kierownik sprzedaży	600000
3	2	Fryderyk	Helc	Sprzedawca	150000
4	2	Zofia	Nowak	Sprzedawca	500000

Prezes Jan Kowalski nie posiada kierownika, więc w kolumnie manager_id występuje wartość NULL. Kierownikiem Zofii Nowak i Fryderyka Helca jest Roman Joświerz, który z kolei podlega Janowi Kowalskiemu.

Za pomocą złączenia własnego możemy wyświetlić imię i nazwisko pracownika oraz jego kierownika. W poniższym zapytaniu występują dwa odwołania do tabeli employees, wykorzystujące aliasy w i m. Alias w jest wykorzystywany do pobrania imienia i nazwiska pracownika, a alias m jest używany do pobrania imienia i nazwiska kierownika. Złączenie własne jest dokonywane za pośrednictwem kolumn w.manager_id i m.employee_id:

```
SELECT w.first_name || ' ' || w.last_name || ' jest podwładnym ' ||
m.first_name || ' ' || m.last_name
FROM employees w, employees m
WHERE w.manager_id = m.employee_id
ORDER BY w.first_name;
```

```
W.FIRST_NAME||' '||W.LAST_NAME||' JEST PODWŁADNYM '||M.FIRST_NAM
```

Fryderyk Helc jest podwładnym Roman Joświerz

Roman Joświerz jest podwładnym Jan Kowalski

Zofia Nowak jest podwładnym Roman Joświerz

Ponieważ w przypadku Jana Kowalskiego `manager_id` ma wartość `NULL`, warunek złączenia nie zwraca tego wiersza.

Dopuszczalne jest łączenie złączeń zewnętrznych i własnych. W poniższym zapytaniu takie połączenie zostało użyte w celu pobrania wiersza dla Jana Kowalskiego. Należy zwrócić uwagę na zastosowanie funkcji `NVL()` w celu wyświetlenia informacji wskazującej, że Jan Kowalski pracuje dla akcjonariuszy (jest prezesem):

```
SELECT w.last_name || ' jest podwładnym ' ||
NVL(m.last_name, 'akcjonariuszy')
FROM employees w, employees m
WHERE w.manager_id = m.employee_id (+)
ORDER BY w.last_name;
```

```
W.LAST_NAME || 'JESTPODWŁADNYM' || NVL(M.LAST
```

```
-----  
Helc jest podwładnym Joświerz  
Joświerz jest podwładnym Kowalski  
Kowalski jest podwładnym akcjonariuszy  
Nowak jest podwładnym Joświerz
```

Wykonywanie złączeń za pomocą składni SQL/92

W prezentowanych dotychczas złączeniach była wykorzystywana składnia Oracle, oparta na standardzie ANSI SQL/86. Od wersji 9g Oracle Database obsługuje standard składni ANSI SQL/92 dla złączeń i należy go stosować w zapytaniach. W tym podrozdziale dowiemy się, jak korzystać z tej składni oraz uniknąć niechcianych iloczynów kartezjańskich.

Wykonywanie złączeń wewnętrznych dwóch tabel z wykorzystaniem składni SQL/92

Wcześniej do wykonania złączenia wewnętrzne stosowaliśmy poniższe zapytanie, zgodne ze standardem SQL/86:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

W standardzie SQL/92 do wykonywania złączeń wewnętrznych służą klauzule `INNER JOIN` i `ON`. Poniższe zapytanie ma takie samo znaczenie jak zapytanie przedstawione powyżej, użycie w nim jednak klauzuli `INNER JOIN` i `ON`:

```
SELECT p.name, pt.name
FROM products p INNER JOIN product_types pt
ON p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

W klauzuli `ON` można również stosować operatory nierówności. Zapytanie to było wcześniej prezentowane z wykorzystaniem nierównozłączenia z użyciem składni SQL/86:

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e, salary_grades sg
WHERE e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

Poniższe zapytanie ma takie samo znaczenie, ale wykorzystuje standard SQL/92

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e INNER JOIN salary_grades sg
ON e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

Upraszczanie złączeń za pomocą słowa kluczowego USING

Standard SQL/92 pozwala jeszcze bardziej uprościć warunek złączenia przez zastosowanie słowa kluczowego USING. Występują tutaj jednak pewne ograniczenia:

- zapytanie musi wykorzystywać równozłączenie,
- kolumny w równozłączaniu muszą mieć taką samą nazwę.

Równozłączenia stanowią większość wykonywanych złączeń, a jeżeli będziemy zawsze nazywali klucze obce tak jak odpowiednie klucze główne, warunki te będą spełnione.

W poniższym zapytaniu wykorzystano słowo kluczowe USING zamiast ON:

```
SELECT p.name, pt.name
FROM products p INNER JOIN product_types pt
USING (product_type_id);
```

Gdybyśmy chcieli pobrać product_type_id, w klauzuli SELECT wystarczyłoby tylko podać nazwę kolumny bez nazwy tabeli lub aliasu:

```
SELECT p.name, pt.name, product_type_id
FROM products p INNER JOIN product_types pt
USING (product_type_id);
```

Jeżeli spróbujemy podać alias tabeli z nazwą kolumny, na przykład p.product_type_id, otrzymamy komunikat o błędzie:

```
SQL> SELECT p.name, pt.name, p.product_type_id
  2  FROM products p INNER JOIN product_types pt
  3  USING (product_type_id);
SELECT p.name, pt.name, p.product_type_id
          *
```

BŁĄD w linii 1:
ORA-25154: część dotycząca kolumn w klauzuli USING nie może mieć kwalifikatora

W klauzuli USING również należy używać jedynie nazwy kolumny. Jeżeli w poprzednim zapytaniu określmy USING (p.product_type_id) zamiast USING (product_type_id), zostanie zgłoszony błąd:

```
SQL> SELECT p.name, pt.name, p.product_type_id
  2  FROM products p INNER JOIN product_types pt
  3  USING (p.product_type_id);
          *
USING (p.product_type_id)
          *
```

BŁĄD w linii 3:
ORA-01748: w tym miejscu dopuszczalne są tylko proste nazwy kolumn



Nie należy używać nazw tabel ani aliasów w odwołaniach do kolumn używanych w klauzuli USING. Powoduje to wystąpienie błędu.

Wykonywanie złączeń wewnętrznych obejmujących więcej niż dwie tabele (SQL/92)

Wcześniej opracowaliśmy następujące zapytanie, pobierające wiersz z tabel customers, purchases, products i product_types:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c, purchases pr, products p, product_types pt
WHERE c.customer_id = pr.customer_id
AND p.product_id = pr.product_id
AND p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

Poniższe zapytanie ma takie samo znaczenie, ale wykorzystuje składnię SQL/92. Należy zwrócić uwagę na sposób wykorzystania relacji kluczów obcych za pomocą klauzul INNER JOIN i USING:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c INNER JOIN purchases pr
USING (customer_id)
INNER JOIN products p
USING (product_id)
INNER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

Wykonywanie złączeń wewnętrznych z użyciem wielu kolumn (SQL/92)

Jeżeli w złączeniu wykorzystujemy kilka kolumn z dwóch tabel, określamy je w klauzuli ON i łączymy za pomocą operatora AND. Założmy, że mamy dwie tabele: tabela1 i tabela2, i chcemy je złączyć, wykorzystując kolumny kolumna1 i kolumna2 z obu tabel. Zapytanie będzie miało postać:

```
SELECT...
FROM tabela1 INNER JOIN tabela2
ON tabela1.kolumna1 = tabela2.kolumna1
AND tabela1.kolumna2 = tabela2.kolumna2;
```

Mögemy jeszcze bardziej uprościć to zapytanie, stosując słowo kluczowe USING, ale tylko wtedy, gdy tworzymy równozłączenie, a nazwy kolumn są identyczne. Poniższe zapytanie ma takie samo znaczenie, wykorzystano w nim jednak słowo kluczowe USING:

```
SELECT...
FROM tabela1 INNER JOIN tabela2
USING (kolumna1, kolumna2);
```

Wykonywanie złączeń zewnętrznych z użyciem składni SQL/92

Wcześniej tworzyliśmy złączenia zewnętrzne wykorzystujące operator złączenia zewnętrznego (+), co stanowi własnościową składnię firmy Oracle. W SQL/92 składnia złączeń zewnętrznych wygląda inaczej. Zamiast operatora (+) określamy typ złączenia w klauzuli FROM, korzystając z następującej składni:

```
FROM tabela1 { LEFT | RIGHT | FULL } OUTER JOIN tabela2
```

gdzie:

- tabela1 i tabela2 są tabelami, które chcemy złączyć,
- LEFT oznacza, że chcemy wykonać złączenie lewostronne,
- RIGHT oznacza, że chcemy wykonać złączenie prawostronne,
- FULL oznacza, że chcemy wykonać pełne złączenie zewnętrzne, które wykorzystuje wszystkie wiersze z tabela1 i tabela2, łącznie z tymi posiadającymi wartości NULL w kolumnach używanych w złączeniu. Za pomocą operatora (+) nie można bezpośrednio utworzyć pełnego złączenia zewnętrznego.

Z kolejnych podrozdziałów dowiesz się, jak tworzyć lewostronne, prawostronne i pełne złączenia zewnętrzne za pomocą składni SQL/92.

Wykonywanie lewostronnych złączeń zewnętrznych z użyciem składni SQL/92

Wcześniej utworzyliśmy zapytanie wykonujące lewostronne złączenie zewnętrzne z użyciem własnościowego operatora Oracle (+):

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id (+)
ORDER BY p.name;
```

Poniższy przykład ma takie samo znaczenie, wykorzystuje jednak słowa kluczowe LEFT OUTER JOIN ze standardu SQL/92:

```
SELECT p.name, pt.name
FROM products p LEFT OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

Wykonywanie prawostronnych złączeń zewnętrznych z użyciem składni SQL/92

Wcześniej utworzyliśmy zapytanie wykonujące prawostronne złączenie zewnętrzne z użyciem własnościowego operatora Oracle (+):

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id (+) = pt.product_type_id
ORDER BY p.name;
```

Poniższy przykład ma takie samo znaczenie, wykorzystuje jednak słowa kluczowe RIGHT OUTER JOIN ze standardu SQL/92:

```
SELECT p.name, pt.name
FROM products p RIGHT OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

Wykonywanie pełnych złączeń zewnętrznych z użyciem składni SQL/92

Pełne złączenie zewnętrzne wykorzystuje wszystkie wiersze złączanych tabel, łącznie z tymi, które zawierają wartość NULL w dowolnej kolumnie wykorzystywanej w złączeniu. Poniższy przykład przedstawia zapytanie, w którym użyto słów kluczowych FULL OUTER JOIN ze standardu SQL/92:

```
SELECT p.name, pt.name
FROM products p FULL OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

NAME	NAME
2412: Powrót	VHS
Chemia	Książka
Muzyka klasyczna	CD
Nauka współczesna	Książka
Pierwsza linia	
Pop 3	CD
Space Force 9	DVD
Supernowa	VHS
Twórczy wrzask	CD
Wojny czołgów	VHS
Z Files	VHS
Z innej planety	DVD
	Czasopismo

Należy zauważać, że zwrócony został zarówno wiersz „Pierwsza linia” z tabeli products, jak i „Czasopismo” z tabeli product_types. Są to wiersze zawierające wartość NULL.

Wykonywanie złączeń własnych z użyciem składni SQL/92

Poniższy przykład wykorzystuje składnię SQL/86 do utworzenia złączenia własnego na tabeli employees:

```
SELECT w.last_name || ' jest podwładnym ' || m.last_name
FROM employees w, employees m
WHERE w.manager_id = m.employee_id;
```

Poniższe zapytanie ma takie samo znaczenie, wykorzystuje jednak składnię SQL/92:

```
SELECT w.last_name || ' jest podwładnym ' || m.last_name
FROM employees w INNER JOIN employees m
ON w.manager_id = m.employee_id;
```

Wykonywanie złączeń krzyżowych z użyciem składni SQL/92

Widzieliśmy wcześniej, że pominięcie warunku złączenia między dwiema tabelami powoduje powstanie iloczynu kartezjańskiego. Korzystając ze składni SQL/92, nie można przypadkowo popełnić takiego błędu, ponieważ w celu utworzenia złączenia zawsze należy określić klauzulę `ON` lub `USING`.

Jeżeli jednak naprawdę chcemy uzyskać iloczyn kartezjański, w standardzie SQL/92 musimy jawnie określić to w zapytaniu za pomocą słów kluczowych `CROSS JOIN`. W poniższym zapytaniu iloczyn kartezjański tabel `product_types` i `products` jest generowany za pomocą tych słów kluczowych:

```
SELECT *
FROM product_types CROSS JOIN products;
```

Podsumowanie

Z tego rozdziału dowiedziałeś się, jak:

- tworzyć zapytania korzystające z jednej i wielu tabel,
- wybrać wszystkie kolumny z tabeli, korzystając z gwiazdki (*) w zapytaniu,
- wykonywać obliczenia arytmetyczne w SQL,
- używać operatorów dodawania i odejmowania z datami,
- odwoływać się do tabel i kolumn za pomocą aliasów,
- łączyć wyniki kolumn za pomocą operatora konkatenacji (||),
- wartości `NULL` są stosowane do reprezentacji nieznanych wartości,
- wyświetlać odrębne wiersze za pomocą operatora `DISTINCT`,
- ograniczyć liczbę pobieranych wierszy przez zastosowanie klauzuli `WHERE`,
- sortować wiersze za pomocą klauzuli `ORDER BY`,
- wykonywać wewnętrzne, zewnętrzne i własne złączenia, korzystając ze składni SQL/86 i SQL/92.

W kolejnym rozdziale zostanie dokładnie omówiony program SQL*Plus.

ROZDZIAŁ

3

SQL*Plus

Z tego rozdziału dowiesz się, jak:

- przeglądać strukturę tabeli,
- edytować instrukcje SQL,
- zapisywać i uruchamiać skrypty zawierające instrukcje SQL i polecenia SQL*Plus,
- formatować wyniki zwracane przez SQL*Plus,
- używać zmiennych w SQL*Plus,
- tworzyć proste raporty,
- uzyskać pomoc z SQL*Plus,
- automatycznie generować instrukcje SQL,
- odłączyć się od bazy danych i zakończyć pracę SQL*Plus.

Przeglądanie struktury tabeli

Znajomość struktury tabeli jest przydatna, ponieważ na podstawie tych informacji możemy utworzyć instrukcję SQL i na przykład określić nazwy kolumn, z których chcemy pobrać dane w zapytaniu. Do przeglądania struktury tabeli służy polecenie DESCRIBE.

W poniższym przykładzie użyto polecenia DESCRIBE do przejrzenia struktury tabeli customers. Należy zwrócić uwagę na brak średnika (;) na końcu polecenia:

```
SQL> DESCRIBE customers
Name          Null?    Type
----- -----
CUSTOMER_ID   NOT NULL NUMBER(38)
FIRST_NAME    NOT NULL VARCHAR2(10)
LAST_NAME     NOT NULL VARCHAR2(10)
DOB           DATE
PHONE         VARCHAR2(12)
```

Wynik polecenia DESCRIBE zawiera trzy kolumny przedstawiające strukturę tabeli:

- **Name** zawiera listę nazw kolumn tabeli. W powyższym przykładzie tabela customers zawiera pięć kolumn: customer_id, first_name, last_name, dob i phone.
- **Null?** określa, czy kolumna może przechowywać wartości NULL. Jeżeli jest wyświetlana wartość NOT NULL, kolumna nie może przechowywać wartości NULL. Jeżeli nie jest wyświetlana żadna wartość, w kolumnie mogą być przechowywane wartości NULL. W powyższym przykładzie w kolumnach customer_id, first_name i last_name nie mogą być przechowywane wartości NULL, a w kolumnach dob i phone mogą.

- **Type** określa typ kolumny. W powyższym przykładzie kolumna `customer_id` jest typu `NUMBER(38)`, a `first_name` — typu `VARCHAR2(10)`.

Możemy oszczędzić sobie nieco wpisywania, stosując skróconą wersję polecenia — `DESC`. Wykorzystano ją w poniższym przykładzie:

```
SQL> DESC products
Name          NULL?    Type
-----        -----
PRODUCT_ID    NOT NULL NUMBER(38)
PRODUCT_TYPE_ID NUMBER(38)
NAME          NOT NULL VARCHAR2(30)
DESCRIPTION   VARCHAR2(50)
PRICE         NUMBER(5,2)
```

Edycja instrukcji SQL

Wpisywane w SQL*Plus tych samych instrukcji SQL staje się nuziące, dlatego pociesząca jest informacja, że program przechowuje w buforze wcześniejszą instrukcję SQL i jest możliwa edycja tworzących ją wierszy.

W tabeli 3.1 wymieniono kilka poleceń służących do edycji instrukcji SQL. W nawiasach kwadratowych umieszczone opcjonalne fragmenty nazw polecień (na przykład polecenie `APPEND` można skrócić do `A`).

Tabela 3.1. Polecenia służące do edycji instrukcji SQL w programie SQL*Plus

Polecenie	Opis
<code>A[PPEND] tekst</code>	Dołącza <i>tekst</i> do bieżącego wiersza
<code>C[CHANGE] /stary/nowy</code>	Zmienia tekst określany przez <i>stary</i> na tekst <i>nowy</i>
<code>CL[EAR] BUFF[ER]</code>	Usuwa wszystkie wiersze z bufora
<code>DEL</code>	Usuwa bieżący wiersz
<code>DEL x</code>	Usuwa wiersz określany przez liczbę <i>x</i> (wiersze są numerowane od 1)
<code>L[IST]</code>	Wyświetla listę wszystkich wierszy przechowywanych w buforze
<code>L[IST] x</code>	Wyświetla wiersz nr <i>x</i>
<code>R[UN] lub /</code>	Uruchamia instrukcję przechowywaną w buforze. Można do tego użyć również polecenia /
<i>x</i>	Wiersz <i>x</i> staje się bieżący

Zajmijmy się teraz przykładami zastosowań polecen edycyjnych programu SQL*Plus. Poniższy przykład przedstawia zapytanie w SQL*Plus:

```
SQL> SELECT customer_id, first_name, last_name
  2  FROM customers
  3  WHERE customer_id = 1;
```

SQL*Plus automatycznie zwiększa numer wiersza po naciśnięciu klawisza *Enter*. Aby uczynić aktywnym wiersz nr 1, należy wpisać 1 w wierszu polecenia:

```
SQL> 1
1* SELECT customer_id, first_name last_name
```

SQL*Plus wyświetla bieżący wiersz oraz jego numer.

W poniższym przykładzie użyto polecenia `APPEND`, aby dodać , dob na końcu wiersza:

```
SQL> APPEND , dob
1* SELECT customer_id, first_name last_name, dob
```

W kolejnym przykładzie użyto polecenia `LIST`, aby wyświetlić wszystkie wiersze przechowywane w buforze:

```
SQL> LIST
1  SELECT customer_id, first_name last_name, dob
```

```
2 FROM customers
3* WHERE customer_id = 1
```

Należy zauważyc, że bieżącym wierszem stał się ostatni, na co wskazuje gwiazdka (*).

W poniższym przykładzie użyto polecenia CHANGE, aby zmienić w ostatnim wierszu customer_id = 1 na customer_id = 2:

```
SQL> CHANGE /customer_id = 1/customer_id = 2
3* WHERE customer_id = 2
```

W kolejnym przykładzie uruchomiono zapytanie za pomocą polecenia RUN:

```
SQL> RUN
1 SELECT customer_id, first_name last_name, dob
2 FROM customers
3* WHERE customer_id = 2
```

```
CUSTOMER_ID LAST_NAME   DOB
----- -----
2 Lidia       68/02/05
```

Aby uruchomić instrukcję SQL, można również wpisać prawy ukośnik (/):

```
SQL> /
CUSTOMER_ID LAST_NAME   DOB
----- -----
2 Lidia       68/02/05
```

Zapisywanie, odczytywanie i uruchamianie plików

SQL*Plus umożliwia zapisywanie, odczytywanie i wykonywanie skryptów zawierających polecenia SQL*Plus i instrukcje SQL. Widzieliśmy już przykład uruchamiania skryptu SQL*Plus: w rozdziale 1. uruchamialiśmy skrypt *store_schema.sql*.

W tabeli 3.2 opisano niektóre polecenia służące do pracy z plikami.

Tabela 3.2. Polecenia służące do pracy z plikami w SQL*Plus

Polecenie	Opis
SAV[E] <i>nazwa_pliku</i> [{REPLACE APPEND }]	Zapisuje zawartość bufora programu SQL*Plus do pliku <i>nazwa_pliku</i> . Polecenie APPEND dodaje zawartość bufora do istniejącego pliku. Polecenie REPLACE nadpisuje istniejący plik
GET <i>nazwa_pliku</i>	Odczytuje do bufora zawartość pliku <i>nazwa_pliku</i>
STA[RT] <i>nazwa_pliku</i>	Odczytuje do bufora zawartość pliku <i>nazwa_pliku</i> , a następnie próbuje wykonać zawartość bufora
@ <i>nazwa_pliku</i>	Działa tak samo jak polecenie START
ED[IT]	Kopiuje zawartość bufora programu SQL*Plus do pliku <i>afiedt.buf</i> , a następnie uruchamia domyślny edytor systemu operacyjnego. Po zakończeniu pracy edytora zawartość edytowanego pliku jest kopiwana do bufora programu
ED[IT] <i>nazwa_pliku</i>	Działa tak samo jak EDIT, ale za pomocą parametru <i>nazwa_pliku</i> można określić plik do edycji
SPO[OL] <i>nazwa_pliku</i>	Kopiuje wyjście programu SQL*Plus do pliku określonego przez <i>nazwa_pliku</i>
SPO[OL] OFF	Przerywa kopianie wyjścia z programu SQL*Plus do pliku, a następnie zamyka go

Przyjrzymy się kilku przykładom zastosowania powyższych polecień SQL*Plus. Aby równolegle wykonywać prezentowane przykłady, należy wpisać zapytanie:

```
SQL> SELECT customer_id, first_name, last_name
2  FROM customers
3  WHERE customer_id = 1;
```

W poniższym przykładzie użyto polecenia `SAVE` w celu zapisania zawartości bufora programu SQL*Plus do pliku o nazwie `cust_query.sql` w katalogu `C:\Pliki_SQL`:

```
SQL> SAVE C:\Pliki_SQL\cust_query.sql
```



Należy utworzyć katalog `Pliki_SQL` przed uruchomieniem powyższego przykładu.

Poniżej to samo polecenie do wykonania w systemie Linux:

```
SQL> SAVE /tmp/cust_query.sql
```

W kolejnym przykładzie użyto polecenia `GET` do wczytania zawartości pliku `cust_query.sql` z katalogu `C:\Pliki_SQL\` w systemie Windows:

```
SQL> GET C:\Pliki_SQL\cust_query.sql
1  SELECT customer_id, first_name, last_name
2  FROM customers
3* WHERE customer_id = 1
```

Poniżej to samo polecenie do wykonania w systemie Linux:

```
SQL> GET /tmp/cust_query.sql
```

W poniższym przykładzie uruchomiono zapytanie za pomocą polecenia `/`:

```
SQL> /
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME
----- -----
1 Jan Nikiel
```

W kolejnym przykładzie użyto polecenia `START` do wczytania i uruchomienia pliku `C:\Pliki_SQL\cust_query.sql` za pomocą jednego polecenia:

```
SQL> START C:\Pliki_SQL\cust_query.sql
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME
----- -----
1 Jan Nikiel
```

Można edytować zawartość buforu programu SQL*Plus, wydając polecenie `EDIT`:

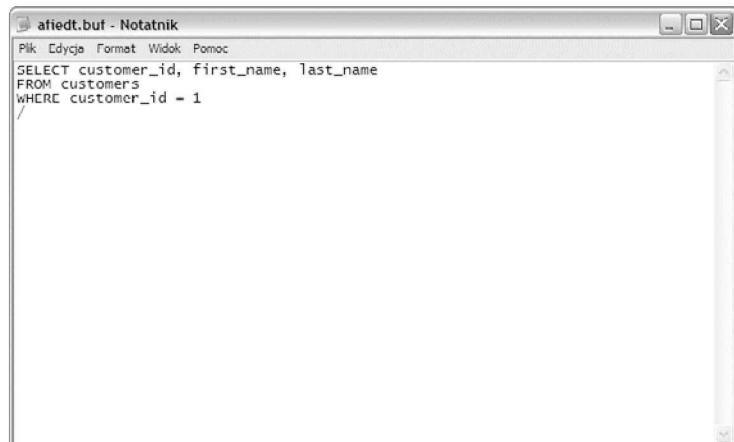
```
SQL> EDIT
```

Polecenie `EDIT` uruchamia domyślny edytor w systemie operacyjnym. W systemie Windows jest to Notatnik. W Uniksach i Linuksach domyślnymi edytorami są odpowiednio vi i emacs.

Rysunek 3.1 przedstawia zawartość bufora programu SQL*Plus w Notatniku. Należy zauważyć, że instrukcja SQL jest zakończona ukośnikiem (/), a nie średnikiem.

Rysunek 3.1.

*Edycja bufora programu
SQL*Plus za pomocą
Notatnika*



Rozwiązywanie problemu z błędem przy próbie skorzystania z edycji

Jeśli przy próbie edycji instrukcji w Windows pojawi się błąd SP2-0110, należy uruchomić SQL*Plus z uprawnieniami administratora. W Windows 7 można to zrobić, klikając prawym klawiszem myszy skrót do SQL*Plus i wybierając *Uruchom jako administrator*. Można ustawić to na stałe. Aby to zrobić, należy kliknąć prawym klawiszem myszy skrót do SQL*Plus, wybrać *Właściwości*, następnie zaznaczyć *Uruchom ten program jako administrator* w zakładce *Zgodność*.

Możesz też ustawić katalog, w którym SQL*Plus ma się uruchamiać. Aby to zrobić, należy kliknąć prawym klawiszem myszy skrót do SQL*Plus, wybrać *Właściwości*, następnie zmienić katalog w polu *Rozpocznij w:* znajdującym się w zakładce *Skrót*. SQL*Plus będzie używał tego domyślnego katalogu do zapisywania i wczytywania plików. Możesz na przykład ustawić ten katalog na C:\Piki_SQL i wtedy SQL*Plus będzie domyślnie zapisywać i wczytywać pliki z tego katalogu.

W edytorze należy zmienić klauzulę WHERE na WHERE customer_id = 2, a następnie zapisać plik i zakończyć pracę edytora (w Notatniku należy wybrać *Plik/Zakończ*, a następnie kliknąć przycisk *Tak*, aby zapisać zapytanie). Aby zapisać plik i zamknąć program edytora ed pracującego w systemie Linux lub Unix, należy wpisać wq.

SQL*Plus wyświetli poniższe wiersze, zawierające zmodyfikowane zapytanie. Należy zauważać, że klauzula WHERE została zmieniona:

```
1 SELECT customer_id, first_name, last_name
2 FROM customers
3* WHERE customer_id = 2
```

Zmienianie domyślnego edytora

Za pomocą polecenia DEFINE w programie SQL*Plus można zmienić domyślny edytor:

```
DEFINE _EDITOR = 'edytor'
```

gdzie *edytor* jest nazwą żądanego edytora.

Na przykład poniższe polecenia ustawia vi jako domyślny edytor:

```
DEFINE EDITOR = 'vi'
```

Dodając wiersz `DEFINE EDITOR 'edytor'` (gdzie *edytor* jest nazwą żądanego edytora) do nowego pliku *login.sql*, można również zmienić domyślny edytor wykorzystywany przez SQL*Plus. W tym pliku można umieszczać dowolne polecenia programu. Podczas uruchamiania sprawdza on, czy w bieżącym katalogu znajduje się ten plik i ewentualnie wykonuje wszystkie zawarte w nim polecenia. Jeżeli w bieżącym katalogu nie ma takiego pliku, SQL*Plus sprawdzi wszystkie katalogi (i ich podkatalogi) określone przez zmienną środowiska SQLPATH.

W systemie Windows jest ona określana przez wpis w rejestrze. Na przykład w systemie Windows zainstalowaną bazą danych Oracle Database 12c zmienna SQLPATH jest ustawiona na E:\oracle_11g\product\12.1.0\dbhome_1\ dbs. Jest to tylko przykładowa ścieżka i może być inna na Twoim komputerze. W systemach Unix i Linux zmienna ta nie jest domyślnie definiowana i należy ją dodać jako zmienianą środowiska.

Więcej informacji na temat konfigurowania pliku *login.sql* można znaleźć w *SQL*Plus User's Guide and Reference* wydanym przez Oracle Corporation.

Do uruchomienia zmodyfikowanego zapytania można użyć polecenia /:

```
SQL> /
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME
2	Lidia	Stal

W celu skopiowania do pliku wyjścia programu SQL*Plus należy użyć polecenia SPOOL. W poniższym przykładzie wyjście jest protokołowane do pliku o nazwie *cust_results.txt*. Następnie zapytanie jest uruchamiane ponownie, po czym protokołowanie wyjścia do pliku zostaje zakończone poleceniem SPOOL OFF:

```
SQL> SPOOL C:\Pliki_SQL\cust_results.txt
SQL> /
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME
-----
2 Lidia Stal
```

```
SQL> SPOOL OFF
```

Plik *cust_results.txt* będzie zawierał wyjście znajdujące się między znakiem / i poleceniem SPOOL OFF.

Formatowanie kolumn

Polecenie COLUMN służy do formatowania nagłówków i danych kolumn. Uproszczona składnia tego polecenia ma następującą postać:

```
COL[UMN] {kolumna | alias} [opcje]
```

gdzie:

- *kolumna* jest nazwą kolumny,
- *alias* jest aliasem kolumny, która będzie formatowana (z rozdziału 2. wiesz, że możemy „zmienić” nazwę kolumny, stosując alias — w poleceniu COLUMN możemy się do takiego aliasu odwołać),
- *opcje* to jedna lub więcej opcji używanych do formatowania kolumny lub aliasu.

Polecenie COLUMN ma wiele opcji. W tabeli 3.3 opisano niektóre z nich.

Tabela 3.3. Opcje polecenia COLUMN

Opcja	Opis
FOR[MAT] <i>format</i>	Ustawia format wyświetlania kolumny lub aliasu zgodnie z napisem <i>format</i>
HEA[DING] <i>nagłówek</i>	Ustawia nagłówek kolumny lub aliasu na napis <i>nagłówek</i>
JUS[TIFY] [{ LEFT CENTER RIGHT }]	Wyrównuje dane kolumny do lewej, do środka lub do prawej
WRA[PPED]	Zawija koniec wpisu do następnego wiersza. Ta opcja może spowodować podzielenie słów na kilka wierszy
WOR[D_WWRAPPED]	Podobna do opcji WRAPPED, ale słowa nie będą zawijane do kilku wierszy
CLE[AR]	Czyści formatowanie kolumn (czyli przywraca domyślne formatowanie)

Napis *format* w powyższej tabeli może przyjmować wiele parametrów formatujących. Przesyłane parametry zależą od rodzaju danych przechowywanych w kolumnie:

- Jeżeli kolumna zawiera znaki, używamy *Ax* do formatowania ich, gdzie *x* określa szerokość pola w znakach. Na przykład A12 ustawia szerokość na 12 znaków.
- Jeżeli kolumna zawiera liczby, możemy użyć jednego z wielu formatów opisanych w tabeli 4.5 w rozdziale 4. Na przykład 99.99C ustawia format na dwie cyfry, po których następuje separator dziesiętny, kolejne dwie cyfry oraz symbol waluty.
- Jeżeli kolumna zawiera daty, można użyć jednego z wielu formatów dat opisanych w tabeli 5.2 w rozdziale 5. Na przykład MM-DD-YYYY ustawia format na dwie cyfry miesiąca (MM), dwie cyfry dnia (DD) i cztery cyfry roku (YYYY).

Zobaczmy, jak formatować wyniki zapytania pobierającego kolumny *product_id*, *name*, *description* i *price* z tabeli *products*. Wymagania dotyczące wyświetlania, napisy formatujące i polecenia COLUMN zostały przedstawione w tabeli 3.4.

Poniższy przykład przedstawia polecenia COLUMN w SQL*Plus:

```
SQL> COLUMN product_id FORMAT 99
SQL> COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WWRAPPED
SQL> COLUMN description FORMAT A13 WORD_WWRAPPED
SQL> COLUMN price FORMAT 99.99C
```

Tabela 3.4. Opis przykładu użycia polecenia COLUMN

Kolumna	Wyświetl jako...	Format	Polecenie COLUMN
product_id	dwie cyfry	99	COLUMN product_id FORMAT 99
name	13-znakowe napisy z zawijaniem słów, nagłówek kolumny ustawiony na PRODUCT_NAME	A13	COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WRAPPED
description	13-znakowe napisy z zawijaniem słów	A13	COLUMN description A13 WORD_WRAPPED
price	dwie cyfry przed i dwie cyfry za separatorem dziesiętnym oraz symbol złotówki	99.99C	COLUMN price FORMAT 99.99C

Poniższy przykład pobiera kilka wierszy z tabeli product. Należy zwrócić uwagę na formatowanie kolumn w wynikach zapytania:

```
SQL> SELECT product_id, name, description, price
  2  FROM products
  3 WHERE product_id < 6;
```

PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	PRICE
1	Nauka współczesna	Opis współczesnej nauki	19.95PLN
2	Chemia	Wprowadzenie do chemii	30.00PLN
3	Supernowa	Eksplozja gwiazdy	25.99PLN
4	Wojny czołgów	Film akcji o nadchodzącej wojnie	13.95PLN
5	Z Files	Serial o tajemniczych zjawiskach	49.99PLN

Wyniki są czytelne, ale czy nie byłoby lepiej, gdyby nagłówki były wyświetlane tylko raz, na samej górze? Jak się przekonamy, należy w tym celu ustawić rozmiar strony.

Ustawianie rozmiaru strony

Polecenie SET PAGESIZE służy do definiowania liczby wierszy na stronie. Ustawia liczbę wierszy, jaką SQL*Plus traktuje jako jedną „stronę” wyników, po której nagłówki zostaną wyświetcone ponownie.

W poniższym przykładzie za pomocą polecenia SET PAGESIZE ustalono rozmiar strony na 100 wierszy i ponownie uruchomiono zapytanie za pomocą ukośnika (/):

```
SQL> SET PAGESIZE 100
SQL> /
```

PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	PRICE
1	Nauka współczesna	Opis współczesnej nauki	19.95PLN
2	Chemia	Wprowadzenie do chemii	30.00PLN

3	Supernowa gwiezdy	Eksplozja	25.99PLN
4	Wojny czołgów	Film akcji o nadchodzącej wojnie	13.95PLN
5	Z Files	Serial o tajemniczych zjawiskach	49.99PLN

Należy zauważać, że nagłówki są wyświetlane tylko raz, na samej górze. Dzięki temu wyniki wyglądają lepiej.



Rozmiar strony może wynosić maksymalnie 50 000 wierszy.

Poniższy przykład przywraca domyślny rozmiar strony (14):

SQL> SET PAGESIZE 14

Ustawianie rozmiaru wiersza

Polecenie SET LINESIZE służy do definiowania liczby znaków w wierszu. W poniższym przykładzie ustwiono rozmiar wiersza na 50 znaków i uruchomiono inne zapytanie:

SQL> SET LINESIZE 50

SQL> SELECT * FROM customers;

```
CUSTOMER_ID FIRST_NAME LAST_NAME   DOB
----- -----
PHONE
-----
1 Jan      Nikiel     65/01/01
800-555-1211

2 Lidia    Stal       68/02/05
800-555-1212

3 Stefan   Brąz       71/03/16
800-555-1213

4 Grażyna Cynk        70/05/20
800-555-1214

5 Jadwiga  Mosiądz   70/05/20
```

Szerokość wierszy nie przekracza 50 znaków.



Szerokość wiersza może wynosić maksymalnie 32 767 znaków.

Poniższy przykład przywraca domyślną szerokość wiersza (80):

SQL> SET LINESIZE 80

Czyszczenie formatowania kolumny

Do czyszczenia formatowania kolumny służy opcja CLEAR polecenia COLUMN. Na przykład poniższe polecenie COLUMN czyści formatowanie kolumny product_id:

SQL> COLUMN product_id CLEAR

Aby wyczyścić formatowanie wszystkich kolumn, należy użyć polecenia `CLEAR COLUMNS`. Na przykład:

```
SQL> CLEAR COLUMNS
```

Po wyczyszczeniu kolumn wyniki zapytań będą wyświetlane z domyślnym formatowaniem.

Używanie zmiennych

Z tego podrozdziału dowiesz się, jak używać zmiennych, które mogą być stosowane w instrukcjach SQL zamiast faktycznych wartości. Takie zmienne nazywamy **zmiennymi podstawianymi**, ponieważ stanowią one substytutły wartości. Po uruchomieniu instrukcji SQL wprowadzamy wartości zmiennych, które później są wstawiane do instrukcji SQL.

Występują dwa typy zmiennych podstawianych:

- **Zmienne tymczasowe.** Zmienna tymczasowa występuje tylko w instrukcji SQL, w której jest używana — nie jest trwała.
- **Zmienne zdefiniowane.** Zmienna zdefiniowana jest przechowywana, dopóki jej jawnie nie usuńemy, nie przedefiniujemy lub nie zakończymy pracy z SQL*Plus.

W tym podrozdziale dowiemy się, jak używać obu typów zmiennych.

Zmienne tymczasowe

Zmienną tymczasową definiujemy przez umieszczenie w instrukcji SQL znaku & i żądanej nazwy zmiennej. Na przykład `&v_product_id` definiuje zmienną o nazwie `v_product_id`.

Po uruchomieniu poniższego zapytania SQL*Plus prosi o wpisanie wartości dla zmiennej `v_product_id` i następnie wykorzystuje ją w klauzuli `WHERE`. Jeżeli jako wartość tej zmiennej wprowadzimy 2, zostaną wyświetcone szczegółowe informacje na temat produktu nr 2.

```
SQL> SELECT product_id, name, price
  2  FROM products
  3 WHERE product_id = &v_product_id;
Enter value for v_product_id: 2
old   3: WHERE product_id = &v_product_id
new   3: WHERE product_id = 2

PRODUCT_ID NAME                  PRICE
----- -----
      2 Chemia                   30
```

Należy zwrócić uwagę, że SQL*Plus:

- prosi o wprowadzenie wartości dla zmiennej `v_product_id`,
- podstawa wprowadzoną wartość w miejsce `v_product_id` w klauzuli `WHERE`.

Program pokazuje podstawienie w wierszach `old` i `new` wyników wraz z numerem wiersza, w którym jest ono dokonywane. W powyższym przykładzie na podstawie wierszy `old` i `new` możemy zauważać, że zmiennej `v_product_set` w klauzuli `WHERE` jest przypisywana wartość 2.

Jeżeli ponownie uruchomimy zapytanie za pomocą ukośnika (/), SQL*Plus poprosi o wprowadzenie nowej wartości dla zmiennej `v_product_id`. Na przykład:

```
SQL> /
Enter value for v_product_id: 3
old   3: WHERE product_id = &v_product_id
new   3: WHERE product_id = 3

PRODUCT_ID NAME                  PRICE
----- -----
      3 Supernowa                25,99
```

SQL*Plus ponownie wypisuje stary wiersz instrukcji SQL (`old 3: WHERE product_id = &v_product_id`), a następnie nowy, zawierający wprowadzoną wartość zmiennej (`new 3: WHERE product_id = 3`).

Dlaczego zmienne są przydatne?

Zmienne są przydatne, ponieważ umożliwiają tworzenie skryptów, które mogą być uruchamiane przez użytkowników nieznających SQL. Skrypt będzie prosił użytkownika o wprowadzenie wartości dla zmiennej, a następnie wykorzystywał ją w instrukcji SQL.

Załóżmy, że chcemy utworzyć skrypt dla użytkownika, który nie zna SQL, ale chce przejrzeć szczegółowe informacje na temat określonego towaru w sklepie. W takiej sytuacji moglibyśmy wpisać na stałe do kodu wartość `product_id` w klauzuli `WHERE` zapytania i umieścić je w skrypcie SQL*Plus. Na przykład poniższe zapytanie pobiera informacje o produkcie nr 1:

```
SELECT product_id, name, price
FROM products
WHERE product_id = 1;
```

Działa, ale pobiera jedynie informacje o produkcie nr 1. Co jeśli chcielibyśmy zmienić wartość `product_id`, aby pobrać inny wiersz? Moglibyśmy zmodyfikować skrypt, ale byłoby to męczące i żmudne. Czyż nie byłoby lepiej, gdybyśmy mogli wprowadzać wartość dla `product_id`? Umożliwia to zastosowanie zmiennej podstawianej.

Sterowanie wypisywaniem wierszy

Za pomocą polecenia `SET VERIFY` możemy sterować wypisywaniem starych i nowych wierszy. Jeżeli wpiszemy `SET VERIFY OFF`, zostanie ono wyłączone, na przykład:

```
SQL> SET VERIFY OFF
SQL> /
Proszę podać wartość dla v_product_id: 4
```

PRODUCT_ID	NAME	PRICE
4	Wojny czołgów	13,95

Aby ponownie włączyć wypisywanie wierszy, należy wpisać `SET VERIFY ON`, na przykład:

```
SQL> SET VERIFY ON
```

Zmianianie znaku definiującego zmienne

Za pomocą polecenia `SET DEFINE` można określić znak definiujący zmenną, inny niż `&`. Poniższy przykład obrazuje, jak ustawić taki znak na `#`, oraz przedstawia nowe zapytanie:

```
SQL> SET DEFINE '#'
SQL> SELECT product_id, name, price
  2  FROM products
  3 WHERE product_id = #v_product_id;
Enter value for v_product_id: 5
old   3: WHERE product_id = #product_id
new   3: WHERE product_id = 5

PRODUCT_ID NAME          PRICE
-----  -----
      5 Z Files        49,99
```

W kolejnym przykładzie użyto polecenia `SET DEFINE` do przywrócenia znaku `&` jako definiującego zmiennej:

```
SQL> SET DEFINE '&'
```

Podstawianie nazw tabel i kolumn za pomocą zmiennych

Zmienne mogą być również używane do podstawiania nazw tabel i kolumn. Na przykład w poniższym zapytaniu zdefiniowano zmienne dla nazwy kolumny (`v_col`), nazwy tabeli (`v_table`) i wartości kolumny (`v_val`):

```
SQL> SELECT name, &v_col
  2  FROM &v_table
```

```

 3 WHERE &v_col = &v_val;
Enter value for v_col: product_type_id
old  1: SELECT name, &v_col
new  1: SELECT name, product_type_id
Enter value for v_table: products
old  2: FROM &v_table
new  2: FROM products
Enter value for v_col: product_type_id
Enter value for v_val: 1
old  3: WHERE &v_col = &v_val
new  3: WHERE product_type_id = 1

```

NAME	PRODUCT_TYPE_ID
Nauka współczesna	1
Chemia	1

Można uniknąć kilkukrotnego wpisywania wartości zmiennej przez zastosowanie `&&`, na przykład:

```

SQL> SELECT name, &&v_col
 2  FROM &v_table
 3 WHERE &&v_col = &v_val;
Enter value for v_col: product_type_id
old  1: SELECT name, &&v_col
new  1: SELECT name, product_type_id
Enter value for v_table: products
old  2: FROM &v_table
new  2: FROM products
Enter value for v_val: 1
old  3: WHERE &&v_col = &v_val
new  3: WHERE product_type_id = 1

```

NAME	PRODUCT_TYPE_ID
Nauka współczesna	1
Chemia	1

Zmienne zapewniają duży stopień swobody podczas pisania zapytań przeznaczonych dla innych użytkowników: można im dostarczyć skrypt wymagający jedynie wpisywania wartości zmiennych.

Zmienne zdefiniowane

Zmienną można zdefiniować przed wykorzystaniem jej w instrukcji SQL i użyć jej wielokrotnie. Zmienna zdefiniowana jest przechowywana, dopóki jej jawnie nie usuniemy, przeddefiniujemy lub do zakończenia pracy z programem SQL*Plus.

Do definiowania zmiennych służy polecenie `DEFINE`. Po zakończeniu pracy ze zmienną można ją usunąć za pomocą polecenia `UNDEFINE`. W tym podrozdziale dokładnie omówię te polecenia. Poznamy również polecenie `ACCEPT`, które umożliwia zdefiniowanie zmiennej i ustawienie jej typu danych.

Zmienne można również definiować w skrypcie SQL*Plus i przesyłać do nich wartości podczas wykonywania go. To umożliwia pisanie pierwotnych raportów, które mogą być uruchamiane przez dowolnych użytkowników, nawet nieznających SQL. Tworzenie prostych raportów zostanie opisane w jednym z kolejnych podrozdziałów.

Definiowanie i wypisywanie zmiennych za pomocą polecenia `DEFINE`

Polecenie `DEFINE` służy zarówno do definiowania nowej zmiennej, jak i wypisywania aktualnie zdefiniowanych. Poniższy przykład definiuje zmienną `v_product_id` i przypisuje jej wartość 7:

```
SQL> DEFINE v_product_id = 7
```

Można przejrzeć definicję zmiennej, wpisując polecenie `DEFINE` oraz nazwę zmiennej. W poniższym przykładzie wyświetlono definicję zmiennej `v_product_id`:

```
SQL> DEFINE v_product_id
DEFINE V_PRODUCT_ID      = "7" (CHAR)
```

Należy zauważyc, że `v_product_id` została zdefiniowana jako zmienna typu CHAR.

Wpisanie samego polecenia `DEFINE` powoduje wyświetlenie wszystkich zmiennych sesji:

```
SQL> DEFINE
DEFINE _DATE          = "14/06/30" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "//localhost/pdborcl" (CHAR)
DEFINE _USER           = "STORE" (CHAR)
DEFINE _PRIVILEGE     = "" (CHAR)
DEFINE _SQLPLUS_RELEASE = "1201000100" (CHAR)
DEFINE _EDITOR         = "Notepad" (CHAR)
DEFINE _O_VERSION      = "Oracle Database 12c Enterprise Edition..." (CHAR)
DEFINE _O_RELEASE       = "1201000100" (CHAR)
DEFINE V_PRODUCT_ID    = "7" (CHAR)
```

Za pomocą zmiennej zdefiniowanej możemy określić w instrukcji SQL wartość kolumny. Na przykład w klauzuli `WHERE` poniższego zapytania zostało zastosowane odwołanie do `v_product_id`:

```
SQL> SELECT product_id, name, price
  2  FROM products
  3 WHERE product_id = &v_product_id;
old   3: WHERE product_id = &v_product_id
new   3: WHERE product_id = 7
```

PRODUCT_ID	NAME	PRICE
7	Space Force 9	13,49

Nie jest wyświetlana prośba o wprowadzenie wartości dla zmiennej `v_product_id`, ponieważ podczas definiowania przypisano jej wartość (7).

Definiowanie i ustawianie zmiennych za pomocą polecenia ACCEPT

Polecenie `ACCEPT` czeka, aż użytkownik wprowadzi wartość dla zmiennej. Może posłużyć do przypisania nowej wartości istniejącej zmiennej lub do zdefiniowania nowej zmiennej i zainicjalizowania jej z wprowadzoną wartością. Umożliwia ono również określenie typu danych dla zmiennej.

Uproszczona składnia tego polecenia ma następującą postać:

```
ACCEPT nazwa_zmiennej [typ] [FORMAT format] [PROMPT monit] [HIDE]
```

gdzie:

- `nazwa_zmiennej` jest nazwą zmiennej.
- `typ` jest typem danych zmiennej. Można użyć typów CHAR, NUMBER i DATE. Domyślnie zmienne są definiowane z typem CHAR. Zmienne typu DATE są tak naprawdę przechowywane jako zmienne CHAR.
- `format` jest formatem używanym dla zmiennej. Przykładem może tu być A15 (15 znaków), 9999 (liczba czterocyfrowa) czy też DD-MON-YYYY (data). Formaty liczb zostały opisane w tabeli 4.5 w rozdziale 4., a formaty dat w tabeli 5.2 w rozdziale 5.
- `monit` jest tekstem wyświetlonym przez SQL*Plus jako prośba o wprowadzenie wartości zmiennej.
- `HIDE` powoduje ukrycie wprowadzanej wartości. Przydaje się to podczas wprowadzania haseł lub innych prywatnych informacji.

Przejdzmy do kilku przykładów użycia polecenia `ACCEPT`. Poniższy definiuje zmienną `v_customer_id` jako liczbę dwucyfrową:

```
SQL> ACCEPT v_customer_id NUMBER FORMAT 99 PROMPT 'Id. klienta: '
Id. klienta: 5
```

Kolejny definiuje zmienną `v_date` typu DATE w formacie DD-MON-YYYY:

```
SQL> ACCEPT v_date DATE FORMAT 'DD-MON-YYYY' PROMPT 'Data: '
Data: 12-gru-2014
```

Następny definiuje zmienną v_password typu CHAR. Wpisywana wartość jest ukrywana, ponieważ została stosowana opcja HIDE:

```
SQL> ACCEPT v_password CHAR PROMPT 'Hasło: ' HIDE
Hasło:
```

W Oracle Database 9i i wcześniejszych wprowadzana wartość jest wyświetlana jako ciąg gwiazdek (*), aby ją ukryć. W Oracle Database 10g i nowszych podczas wpisywania wartości nic nie jest wyświetlane.

Za pomocą polecenia DEFINE można przejrzeć zdefiniowane zmienne:

```
SQL> DEFINE
...
DEFINE v_customer_id      =          5 (NUMBER)
DEFINE v_date              = "12-gru-2006" (CHAR)
DEFINE v_password           = "1234567" (CHAR)
DEFINE v_product_id        = "7" (CHAR)
```

Należy zauważyć, że zmienna v_date jest przechowywana jako typ CHAR.

Usuwanie zmiennych za pomocą polecenia UNDEFINE

Do usuwania zmiennych służy polecenie UNDEFINE. W poniższym przykładzie za pomocą polecenia UNDEFINE usuwane są zmienne v_customer_id, v_date, v_password i v_product_id:

```
SQL> UNDEFINE v_customer_id
SQL> UNDEFINE v_date
SQL> UNDEFINE v_password
SQL> UNDEFINE v_product_id
```

 **Uwaga** Po zakończeniu pracy programu SQL*Plus są usuwane wszystkie zmienne, nawet jeżeli nie zostały jawnie usunięte za pomocą polecenia UNDEFINE.

Tworzenie prostych raportów

Zmienne mogą być wykorzystywane w skryptach SQL*Plus do tworzenia raportów uruchamianych przez użytkowników. Skrypty opisywane w tym podrozdziale znajdują się w paczce z kodami dołączonymi do książki, w katalogu SQL.

 **Wskazówka** SQL*Plus nie został zaprojektowany jako narzędzie raportujące. Jeżeli wymagane są kompleksowe możliwości raportowania, należy użyć Oracle Reports lub podobnych programów.

Używanie zmiennych tymczasowych w skrypcie

Poniższy skrypt *report1.sql* wykorzystuje w klauzuli WHERE zapytania o zmienną tymczasową o nazwie v_product_id:

```
-- wyłącz wyświetlanie instrukcji i komunikatów weryfikacji
SET ECHO OFF
SET VERIFY OFF
```

```
SELECT product_id, name, price
FROM products
WHERE product_id = &v_product_id;
```

Polecenie SET ECHO OFF wyłącza wyświetlanie instrukcji SQL i poleceń skryptu. Polecenie SET VERIFY OFF wyłącza wyświetlanie komunikatów weryfikacji tych dwóch poleceń, aby zminimalizować liczbę dodatkowych wierszy pokazywanych w SQL*Plus podczas wykonywania skryptu.

Skrypt *report1.sql* można uruchomić w SQL*Plus za pomocą polecenia @:

```
SQL> @ C:\sql_book\SQL\report1.sql
Enter value for v_product_id: 2
```

PRODUCT_ID	NAME	PRICE
2	Chemia	30

Należy zastąpić ścieżkę dostępu w przykładzie taką, w której zostały zapisane pliki dla tej książki. Ponadto jeżeli w nazwach katalogów występują spacje, należy umieścić w cudzysłowie cały tekst znajdujący się za poleceniem @, na przykład:

```
@ "C:\mój katalog\sql book\SQL\report1.sql"
```

Używanie zmiennych zdefiniowanych w skrypcie

W poniższym skrypcie, *report2.sql*, wykorzystano polecenie ACCEPT do zdefiniowania zmiennej o nazwie *v_product_id*:

```
SET ECHO OFF
SET VERIFY OFF
```

```
ACCEPT v_product_id NUMBER FORMAT 99 PROMPT 'Id. produktu: '
```

```
SELECT product_id, name, price
FROM products
WHERE product_id = &v_product_id;
```

-- czyszczenie

```
UNDEFINE v_product_id
```

Został zastosowany przyjazny tekst prośby o wprowadzenie wartości *v_product_id*, a zmienna *v_product_id* została usunięta na końcu skryptu.

Skrypt *report2.sql* można uruchomić za pomocą SQL*Plus:

```
SQL> @ C:\sql_book\SQL\report2.sql
Id. produktu: 4
```

PRODUCT_ID	NAME	PRICE
4	Wojny czołgów	13,95

Przesyłanie wartości do zmiennej w skrypcie

Uruchamiając skrypt, można przesyłać wartość zmiennej. Wówczas odwołujemy się do zmiennej w skrypcie za pomocą liczby. Skrypt *report3.sql* jest takim przykładem. Należy zwrócić uwagę, że zmienna jest identyfikowana za pomocą &1:

```
SET ECHO OFF
SET VERIFY OFF
```

```
SELECT product_id, name, price
FROM products
WHERE product_id = &1;
```

Przy uruchamianiu skryptu *report3.sql* należy przesyłać wartość zmiennej za nazwą skryptu. Poniższy przykład przesyła wartość 3 do skryptu *report3.sql*:

```
SQL> @ C:\sql_book\SQL\report3.sql 3
```

PRODUCT_ID	NAME	PRICE
3	Supernowa	25,99

Jeżeli w nazwach katalogów, w których zostały zapisane skrypty, znajdują się spacje, należy umieścić nazwy katalogów i skryptów w cudzysłowie, na przykład:

```
@ "C:\mój katalog\sql book\SQL\report3.sql" 3
```

Do skryptu można przesyłać dowolną liczbę parametrów, przy czym pozycja każdej wartości odpowiada liczbie w skrypcie. Pierwszy parametr odpowiada &1, drugi &2 itd. Skrypt *report4.sql* zawiera dwa parametry:

```
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, product_type_id, name, price
FROM products
WHERE product_type_id = &1
AND price > &2;
```

W poniższym przykładzie do skryptu *report4.sql* przesłano wartości 1 i 9,99 odpowiednio dla wartości zmiennej &1 i &2:

```
SQL> @ C:\sql_book\SQL\report4.sql 1 9.99
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Nauka współczesna	19,95
2	1	Chemia	30

Ponieważ &1 przypisano wartość 1, kolumna *product_type_id* w klauzuli WHERE jest ustawiana na 1, a ponieważ &2 przypisano wartość 9,99, kolumna *price* w klauzuli WHERE jest ustawiana na 9,99. Dlatego też są wyświetlane wiersze z *product_type_id* równym 1 i *price* większym niż 9,99.

Dodawanie nagłówka i stopki

Nagłówek i stopkę raportu możemy dodać za pomocą poleceń TTITLE i BTITLE. Poniżej przedstawiono przykład polecenia TTITLE:

```
TTITLE LEFT 'Sporządzono: ' _DATE CENTER 'Sporządził: ' SQL.USER RIGHT 'Strona: ' FORMAT 999 SQL.PNO SKIP 2
```

Polecenie to zawiera następujące elementy:

- *_DATE* — wyświetla bieżącą datę,
- *SQL.USER* — wyświetla nazwę bieżącego użytkownika,
- *SQL.PNO* — wyświetla bieżącą stronę (FORMAT użyto do sformatowania liczby),
- *LEFT, CENTER i RIGHT* — służą do wyrównania tekstu,
- *SKIP 2* — opuszcza dwa wiersze.

Jeżeli przykładowy skrypt zostałby uruchomiony 12 sierpnia 2012 roku przez użytkownika store, zawierałby następujący nagłówek:

```
Sporządzono: 12/08/12 Sporządził: STORE Strona: 1
```

Kolejny przykład przedstawia polecenie BTITLE:

```
BTITLE CENTER 'Dziękujemy za sporządzenie raportu' RIGHT 'Strona: ' FORMAT 999 SQL.PNO
```

To polecenie wyświetla:

```
Dziękujemy za sporządzenie raportu Strona: 1
```

Skrypt *report5.sql* zawiera przedstawione wyżej polecenia TTITLE i BTITLE:

```
TTITLE 'Sporządzono: ' _DATE CENTER 'Sporządził: ' SQL.USER RIGHT 'Strona: ' FORMAT 999 SQL.PNO SKIP 2
BTITLE CENTER 'Dziękujemy za sporządzenie raportu' RIGHT 'Strona: ' FORMAT 999 SQL.PNO
```

```
SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 30
SET LINESIZE 70
CLEAR COLUMNS
COLUMN product_id HEADING ID FORMAT 99
COLUMN name HEADING 'Nazwa produktu' FORMAT A20 WORD_WWRAPPED
```

```
COLUMN description HEADING Opis FORMAT A30 WORD_WRAPPED
COLUMN price HEADING Cena FORMAT 99.99C
```

```
SELECT product_id, name, description, price
FROM products;
```

```
CLEAR COLUMNS
TTITLE OFF
BTITLE OFF
```

Ostatnie dwa wiersze wyłączają nagłówek i stopkę zdefiniowane przez polecenia TTITLE i BTITLE.

Poniższy przykład prezentuje wynik uruchomienia skryptu *report5.sql*:

```
SQL> @ C:\sql_book\SQL\report5.sql
```

Sporządzono: 12/09/15 Sporządził: STORE Strona: 1

ID	Nazwa produktu	Opis	Cena
1	Nauka współczesna	Opis współczesnej nauki	19.95PLN
2	Chemia	Wprowadzenie do chemii	30.00PLN
3	Supernowa	Eksplozja gwiazdy	25.99PLN
4	Wojny czołgów	Film akcji o nadchodzącej wojnie	13.95PLN
5	Z Files	Serial o tajemniczych zjawiskach	49.99PLN
6	2412: Powrót	Powrót obcych	14.95PLN
7	Space Force 9	Przygody bohaterów	13.49PLN
8	Z innej planety	Obcy z innej planety ląduje na Ziemi	12.99PLN
9	Muzyka klasyczna	Nalepsze dzieła muzyki klasycznej	10.99PLN
10	Pop 3	Najlepsze utwory popowe	15.99PLN
11	Twórczy wrzask	Album debiutancki	14.99PLN
12	Pierwsza linia	Największe hity	13.49PLN

Dziękujemy za sporządzenie raportu Strona: 1

Obliczanie sum pośrednich

Za pomocą polecień BREAK ON i COMPUTE możemy dodać sumę pośrednią dla kolumny. Polecenie BREAK ON powoduje wprowadzenie przerwy w wynikach na podstawie zmian wartości w kolumnie, a COMPUTE oblicza wartości w kolumnie.

Skrypt *report6.sql* prezentuje, jak można obliczyć sumy pośrednie dla produktów tego samego typu:

```
BREAK ON product_type_id
COMPUTE SUM OF price ON product_type_id
```

```
SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 50
SET LINESIZE 70
```

```
CLEAR COLUMNS
COLUMN price HEADING Cena FORMAT 999.99C
```

```
SELECT product_type_id, name, price
FROM products
ORDER BY product_type_id;
```

```
CLEAR COLUMNS
```

Poniżej przedstawiono wynik działania skryptu *report6.sql*:

```
SQL> @ C:\sql_book\SQL\report6.sql
```

PRODUCT_TYPE_ID	NAME	Cena
1	Nauka współczesna	19.95PLN
	Chemia	30.00PLN
*****	*****	*****
sum		49.95PLN
2	Z Files	49.99PLN
	Wojny czołgów	13.95PLN
	Supernowa	25.99PLN
	2412: Powrót	14.95PLN
*****	*****	*****
sum		104.88PLN
3	Space Force 9	13.49PLN
	Z innej planety	12.99PLN
*****	*****	*****
sum		26.48PLN
4	Muzyka klasyczna	10.99PLN
	Pop 3	15.99PLN
	Twórczy wrzask	14.99PLN
*****	*****	*****
sum		41.97PLN
	Pierwsza linia	13.49PLN
*****	*****	*****
sum		13.49PLN

Należy zauważyć, że gdy w kolumnie `product_type_id` pojawi się nowa wartość, SQL*Plus przerewa wypisywanie i oblicza sumę wartości z kolumny `price` dla tego samego `product_type_id`. Jego wartość jest wypisywana tylko raz dla wszystkich wierszy z tą wartością, na przykład „Nauka współczesna” i „Chemia” są książkami i w ich przypadku ma on wartość 1, która jest wyświetlana jedynie przy pozycji „Nauka współczesna”. Suma cen tych dwóch książek wynosi 49,95 zł. Pozostałe sekcje raportu zawierają sumy cen produktów z innymi wartościami `product_type_id`.

Uzyskiwanie pomocy od SQL*Plus

Polecenie `HELP` powoduje wyświetlenie pomocy w programie SQL*Plus:

```
SQL> HELP
```

```
HELP
```

```
----
```

```
Accesses this command line help system. Enter HELP INDEX or ? INDEX for a list of topics.
```

```
You can view SQL*Plus resources at  
http://www.oracle.com/technology/documentation/
```

```
HELP|? [topic]
```

W kolejnym przykładzie uruchomiono polecenie `HELP INDEX`:

```
SQL> HELP INDEX
```

Enter Help [topic] for help.

@	COPY	PAUSE	SHUTDOWN
@@	DEFINE	PRINT	SPool
/	DEL	PROMPT	SQLPLUS
ACCEPT	DESCRIBE	QUIT	START
APPEND	DISCONNECT	RECOVER	STARTUP
ARCHIVE LOG	EDIT	REMARK	STORE

ATTRIBUTE	EXECUTE	REPFOOTER	TIMING
BREAK	EXIT	REPHEADER	TTITLE
BTITLE	GET	RESERVED WORDS (SQL)	UNDEFINE
CHANGE	HELP	RESERVED WORDS (PL/SQL)	VARIABLE
CLEAR	HOST	RUN	WHENEVER OSERROR
COLUMN	INPUT	SAVE	WHENEVER SQLERROR
COMPUTE	LIST	SET	XQUERY
CONNECT	PASSWORD	SHOW	

W poniższym przykładzie uruchomiono polecenie HELP EDIT:

SQL> **HELP EDIT**

EDIT

Invokes an operating system text editor on the contents of the specified file or on the contents of the SQL buffer. The buffer has no command history list and does not record SQL*Plus commands.

ED[IT] [file_name[.ext]]

Automatyczne generowanie instrukcji SQL

W tym podrozdziale zostanie krótka opisana technika pisania instrukcji SQL tworzących nowe instrukcje SQL. Ta możliwość jest bardzo przydatna i może nam zaoszczędzić sporo czasu podczas pisania podobnych do siebie instrukcji SQL.

Jednym z prostych przykładów jest instrukcja SQL tworząca instrukcje `DROP TABLE`, usuwające tabele z bazy danych. Poniższe zapytanie tworzy serię instrukcji `DROP TABLE`, usuwających tabele ze schematu store:

```
SELECT 'DROP TABLE ' || table_name || ';' 
FROM user_tables;
```

```
'DROPTABLE'||TABLE_NAME||';'
```

```
-----
```

```
DROP TABLE ALL_SALES;
```

```
DROP TABLE ALL_SALES2;
```

```
...
```

```
DROP TABLE REG_EXPS;
```

```
DROP TABLE SALARY_GRADES;
```

Dla zachowania przejrzystości pominąłem większość wierszy wyniku. Wygenerowane instrukcje SQL można zapisać do pliku i uruchomić później.



Tabela `user_tables` zawiera szczegółowe informacje o tabelach w schemacie użytkownika. Kolumna `table_name` zawiera nazwy tabel.

Kończenie połączenia z bazą danych i pracy SQL*Plus

Aby zakończyć połączenie z bazą danych i pozostawić uruchomiony program SQL*Plus, należy wpisać polecenie `DISCONNECT` (program wykona automatycznie polecenie `COMMIT` przed rozłączeniem).

Podczas połączenia z bazą danych SQL*Plus utrzymuje sesję. Po zakończeniu połączenia sesja jest zamknięta. Za pomocą polecenia `CONNECT` można ponownie połączyć się z bazą danych.

Aby zakończyć działanie programu SQL*Plus, należy wpisać `EXIT` lub `QUIT`.

Podsumowanie

Z tego rozdziału dowiedziałeś się, jak:

- przejrzeć strukturę tabeli,
- edytować instrukcję SQL,
- zapisywać, odczytywać i uruchamiać pliki zawierające instrukcje SQL i polecenia SQL*Plus,
- używać zmiennych w SQL*Plus,
- tworzyć proste raporty,
- pisać instrukcje SQL generujące inne instrukcje SQL.

Dokładniejsze omówienie programu SQL*Plus można znaleźć w *SQL*Plus User's Guide and Reference* opublikowanym przez Oracle Corporation.

W kolejnym rozdziale zostały zawarte informacje o tym, jak korzystać z funkcji.

ROZDZIAŁ

4

Proste funkcje

W tym rozdziale dowiesz się, jak:

- przetwarzać wiersze za pomocą funkcji,
- grupować wiersze w bloki za pomocą klauzuli GROUP BY,
- filtrować grupy wierszy za pomocą klauzuli HAVING.

Typy funkcji

W bazie danych Oracle występują dwa główne typy funkcji:

- **Funkcje jednowierszowe** operują na jednym wierszu i zwracają jeden wiersz wyników dla każdego wiersza na wejściu. Przykładem funkcji jednowierszowej jest `CONCAT(x, y)`, która dołącza `y` do `x` i zwraca powstały napis.
- **Funkcje agregujące** operują na kilku wierszach jednocześnie i zwracają jeden wiersz wyników. Przykładem funkcji agregującej jest `Avg(x)`, która zwraca średnią `x`.

Zacznę od omówienia funkcji jednowierszowych, a następnie przejdziemy do funkcji agregujących. W dalszej części książki zostaną przedstawione bardziej złożone funkcje.

Funkcje jednowierszowe

Funkcja jednowierszowa operuje na jednym wierszu i zwraca jeden wiersz wyników dla każdego wiersza na wejściu. Występuje pięć głównych typów funkcji jednowierszowych:

- **funkcje znakowe** — manipulują napisami,
- **funkcje numeryczne** — wykonują obliczenia,
- **funkcje konwertujące** — konwertują wartość z jednego typu na inny,
- **funkcje dat** — przetwarzają daty i czas,
- **funkcje wyrażeń regularnych** — wykorzystują wyrażenia regularne do wyszukiwania danych; zostały wprowadzone w Oracle Database 10g.

Rozpoczniemy od omówienia funkcji znakowych, a następnie przejdziemy do numerycznych, konwertujących oraz wyrażeń regularnych. Funkcje dat zostaną opisane w następnym rozdziale.

Funkcje znakowe

Funkcje znakowe przyjmują wejście znakowe, które może pochodzić z kolumny tabeli lub z dowolnego wyrażenia. Dane wejściowe są przetwarzane i jest zwracany wynik. Przykładem funkcji znakowej jest `UPPER()`, która zwraca napis wejściowy po przekształceniu na wielkie litery. Innym przykładem jest `NVL()`, która konwertuje wartość `NULL` na inną.

W tabeli 4.1, w której zostały opisane niektóre funkcje znakowe, oraz we wszystkich kolejnych definicjach składni *x* i *y* mogą reprezentować kolumny tabeli lub dowolne poprawne wyrażenie. W kolejnych podrozdziałach zostaną dokładniej opisane funkcje wymienione w tabeli 4.1.

Tabela 4.1. Funkcje znakowe

Funkcja	Opis
<code>ASCII(x)</code>	Zwraca kod ASCII znaku <i>x</i>
<code>CHR(x)</code>	Zwraca znak o kodzie ASCII <i>x</i>
<code>CONCAT(x, y)</code>	Dołączyc <i>y</i> do <i>x</i> i zwraca powstały napis
<code>INITCAP(x)</code>	Przekształca pierwszą literę każdego słowa w <i>x</i> na wielką i zwraca nowy napis
<code>INSTR(x, szukany_napis [, start] [, wystąpienie])</code>	Wyszukuje w <i>x</i> napis <i>szukany_napis</i> i zwraca pozycję, w której on występuje. Można przesłać opcjonalne parametry: <ul style="list-style-type: none">■ <i>start</i> — określający pozycję, od której rozpocznie się wyszukiwanie. Pierwszy znak w <i>x</i> ma pozycję 1. Wartość <i>start</i> może być dodatnia lub ujemna. Wartość dodatnia określa pozycję liczoną od początku <i>x</i>. Wartość ujemna określa pozycję liczoną od końca <i>x</i>.■ <i>wystąpienie</i> — określający, które wystąpienie <i>szukany_napis</i> zostanie zwrócone.
<code>LENGTH(x)</code>	Zwraca liczbę znaków w <i>x</i>
<code>LOWER(x)</code>	Przekształca litery w <i>x</i> na małe i zwraca nowy napis
<code>LPAD(x, szerokość, [napis_dopełnienia])</code>	Dopełnia <i>x</i> znakami spacji po lewej stronie, aby uzyskać całkowitą długość napisu równą <i>szerokość</i> . Można przesłać opcjonalny parametr <i>napis_dopełnienia</i> , określający napis, który będzie powtarzany po lewej stronie <i>x</i> w celu wypełnienia dopełnianego obszaru. Zwracany jest dopełniony napis
<code>LTRIM (x [, napis_przycinany])</code>	Usuwa znaki znajdujące się po lewej stronie <i>x</i> . Można przesłać opcjonalny parametr <i>napis_przycinany</i> , określający znaki, które zostaną usunięte. Jeżeli ten parametr nie zostanie przesłany, domyślnie usuwane będą znaki spacji
<code>NANVL(x, wartość)</code>	Zwraca <i>wartość</i> , jeżeli <i>x</i> jest wartością specjalną <code>NAN</code> (nieliczba). W przeciwnym wypadku zwracana jest wartość <i>x</i> . (Ta funkcja została wprowadzona w Oracle Database 10g).
<code>NVL(x, wartość)</code>	Zwraca <i>wartość</i> , jeżeli <i>x</i> to <code>NULL</code> . W przeciwnym razie zwraca <i>x</i>
<code>NVL2(x, wartość1, wartość2)</code>	Zwraca <i>wartość1</i> , jeżeli <i>x</i> to nie <code>NULL</code> . W przeciwnym razie zwraca <i>wartość2</i>
<code>REPLACE(x, szukany_napis, napis_zastępujący)</code>	Wyszukuje w <i>x</i> napis <i>szukany_napis</i> i zastępuje go napisem <i>napis_zastępujący</i>
<code>RPAD(x, szerokość [, napis_dopełnienia])</code>	Dopełnia <i>x</i> znakami spacji po prawej stronie, aby uzyskać całkowitą długość napisu równą <i>szerokość</i> . Można przesłać opcjonalny parametr <i>napis_dopełnienia</i> , określający napis, który będzie powtarzany po lewej stronie <i>x</i> w celu wypełnienia dopełnianego obszaru. Zwracany jest dopełniony napis.

Tabela 4.1. Funkcje znakowe — ciąg dalszy

Funkcja	Opis
RTRIM(x, [, napis_przycinany])	Usuwa znaki znajdujące się po prawej stronie x. Można przesyłać opcjonalny parametr <i>napis_przycinany</i> , określający znaki, które zostaną usunięte. Jeżeli ten parametr nie zostanie przesłany, domyślnie usuwane będą znaki spacji.
SOUNDEX(x)	Zwraca napis zawierający fonetyczną reprezentację x. To umożliwia porównywanie słów, które podobnie brzmią w języku angielskim, ale ich pisownia jest inna
SUBSTR(x, start [, długość])	Zwraca podnapis napisu x, rozpoczynający się w pozycji określonej przez <i>start</i> . Pierwszy znak w x ma pozycję 1. Wartość <i>start</i> może być dodatnia lub ujemna. Wartość dodatnia określa pozycję liczoną od początku x. Wartość ujemna określa pozycję liczoną od końca x. Można przesyłać opcjonalny parametr <i>długość</i> , określający długość podnapisu
TRIM([usuwany_znak FROM] x)	Usuwa znaki po obu stronach x. Można przesyłać opcjonalny parametr <i>usuwany_znak</i> , określający znak do usunięcia. Jeżeli parametr ten nie zostanie przesłany, domyślnie zostaną usunięte znaki spacji
UPPER(x)	Zmienia litery w x na wielkie i zwraca nowy napis

ASCII() i CHR()

Funkcja ASCII(x) zwraca kod ASCII znaku x. Funkcja CHR(x) zwraca znak o kodzie ASCII x.

Poniższe zapytanie pobiera za pomocą funkcji ASCII() kody ASCII znaków a, A, z, Z, 0 i 9:

```
SELECT ASCII('a'), ASCII('A'), ASCII('z'), ASCII('Z'), ASCII(0), ASCII(9)
FROM dual;
```

ASCII('A')	ASCII('A')	ASCII('Z')	ASCII('Z')	ASCII(0)	ASCII(9)
-----	-----	-----	-----	-----	-----
97	65	122	90	48	57



W tym zapytaniu wykorzystano tabelę dual. Zawiera ona jeden wiersz, za pomocą którego możemy wykonywać zapytania niewykorzystujące żadnej konkretnej tabeli.

Poniższe zapytanie pobiera za pomocą funkcji CHR() znaki o kodach ASCII 97, 65, 122, 90, 48 i 57:

```
SELECT CHR(97), CHR(65), CHR(122), CHR(90), CHR(48), CHR(57)
FROM dual;
```

```
C C C C C C
- - - - -
a A z Z 0 9
```

Znaki zwrócone przez funkcję CHR() są tymi samymi, które były przesyłane do funkcji ASCII() w poprzednim zapytaniu. Funkcje CHR() i ASCII() mają zatem przeciwnie działanie.

CONCAT()

Funkcja CONCAT(x, y) dołącza y do x i zwraca nowy napis.

Poniższe zapytanie dołącza za pomocą funkcji CONCAT() wartość z kolumny *last_name* do wartości z kolumny *first_name*:

```
SELECT CONCAT(first_name, last_name)
FROM customers;
```

```
CONCAT(FIRST_NAME,LA
-----
```

JanNikiel
 LidiaStal
 StefanBrąz
 GrażynaCynk
 JadwigaMosiądz



Działanie funkcji `CONCAT()` jest takie samo jak operatora `||`, który został opisany w rozdziale 2.

Uwaga

INITCAP()

Funkcja `INITCAP(x)` zmienia pierwszą literę każdego słowa w `x` na wielką.

Poniższe zapytanie pobiera kolumny `product_id` i `description` z tabeli `products`, a następnie za pomocą funkcji `INITCAP()` zmienia pierwszą literę każdego słowa w `description` na wielką:

```
SELECT product_id, INITCAP(description)
FROM products
WHERE product_id < 4;

PRODUCT_ID INITCAP(DESCRIPTION)
-----
1 Opis Współczesnej Nauki
2 Wprowadzenie Do Chemicznych
   Zjawisk Fizycznych
3 Eksplozja Gwiazdy
```

INSTR()

Funkcja `INSTR(x, szukany_napis [, start] [, wystąpienie])` wyszukuje w `x` napis `szukany_napis` i zwraca pozycję, na której się on znajduje. Można przesłać opcjonalne parametry:

- `start` — określający pozycję rozpoczęcia wyszukiwania. Pierwszy znak w `x` ma pozycję 1. Wartość `start` może być dodatnia lub ujemna. Wartość dodatnia określa pozycję liczoną od początku `x`. Wartość ujemna określa pozycję liczoną od końca `x`.
- `wystąpienie` — określający, które wystąpienie napisu `szukany_napis` zostanie zwrócone.

Poniższe zapytanie pobiera pozycję, na której znajduje się napis `współczesna` w kolumnie `name` pierwszego produktu:

```
SELECT name, INSTR(name, 'współczesna')
FROM products
WHERE product_id = 1;

NAME           INSTR(NAME, 'WSPÓŁCZESNA')
-----
Nauka współczesna          7
```

Kolejne zapytanie wyświetla pozycję, na której znajduje się drugie wystąpienie znaku `a`, rozpoczynając od początku nazwy produktu:

```
SELECT name, INSTR(name, 'a', 1, 2)
FROM products
WHERE product_id = 1;

NAME           INSTR(NAME, 'A', 1, 2)
-----
Nauka współczesna          5
```

Drugie „`a`” w tytule „`Nauka współczesna`” znajduje się na piątej pozycji.

Kolejne zapytanie wyświetla pozycję, na której znajduje się trzecie wystąpienie znaku `a`, licząc od końca nazwy produktu:

```
SELECT name, INSTR(name, 'a', -1, 3)
FROM products
WHERE product_id = 1;
```

NAME	INSTR(NAME, 'A', -1, 3)
------	-------------------------

Nauka współczesna	2
-------------------	---

Trzecie „a” w tytule „Nauka współczesna” znajduje się na drugiej pozycji.

Funkcje znakowe mogą również operować na datach. Poniższe zapytanie pobiera pozycję, na której znajduje się napis STY w kolumnie dob klienta nr 1:

```
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';
SELECT customer_id, dob, INSTR(dob, 'STY')
FROM customers
WHERE customer_id = 1;
```

CUSTOMER_ID	DOB	INSTR(DOB, 'STY')
-------------	-----	-------------------

1	01-STY-65	4
---	-----------	---

LENGTH()

Funkcja LENGTH(*x*) zwraca liczbę znaków w *x*. Poniższe zapytanie za pomocą tej funkcji pobiera długość napisów w kolumnie name tabeli products:

```
SELECT name, LENGTH(name)
FROM products;
```

NAME	LENGTH(NAME)
------	--------------

Nauka współczesna	17
Chemia	6
Supernowa	9
Wojny czołgów	13
Z Files	7
2412: Powrót	12
Space Force 9	13
Z innej planety	15
Muzyka klasyczna	16
Pop 3	5
Twórczy wrzask	14
Pierwsza linia	14

Kolejne zapytanie pobiera całkowitą liczbę znaków składających się na cenę produktu (kolumna price). Należy zwrócić uwagę, że separator dziesiętny (,) jest liczony jako znak w kolumnie price:

```
SELECT price, LENGTH(price)
FROM products
WHERE product_id < 3;
```

PRICE	LENGTH(PRICE)
-------	---------------

19,95	5
30	2

LOWER() i UPPER()

Funkcja LOWER(*x*) zmienia litery w *x* na małe. Funkcja UPPER(*x*) zmienia natomiast litery w *x* na wielkie.

Poniższe zapytanie zmienia litery w napisach z kolumny first_name na wielkie, a litery z napisów z kolumny last_name na małe:

```
SELECT UPPER(first_name), LOWER(last_name)
FROM customers;
```

UPPER(FIRST_NAME)	LOWER(LAST_NAME)
-------------------	------------------

JAN	nikiel
LIDIA	stal

STEFAN	brąz
GRAŻYNA	cynk
JADWIGA	mosiądz

LPAD() i RPAD()

Funkcje LPAD() i RPAD() można opisać w dwóch punktach:

- LPAD(*x*, *szerokość*, [*napis_dopełnienia*]) dopełnia lewą stronę *x* znakami spacji, aby uzupełnić długość napisu *x* do *szerokość* znaków. Można przesyłać opcjonalny parametr *napis_dopełnienia*, który określa napis powtarzany po lewej stronie napisu *x* w celu dopełnienia go. Zwracany jest dopełniony łańcuch.
- RPAD(*x*, *szerokość*, [*napis_dopełnienia*]) dopełnia prawą stronę napisu *x*.

Poniższe zapytanie pobiera kolumny *name* i *price* z tabeli *products*. Kolumna *name* jest dopełniana po prawej stronie za pomocą funkcji RPAD() do długości 30 znaków. Dopełnienie jest wypełniane kropkami. Kolumna *price* jest dopełniana po lewej stronie za pomocą funkcji LPAD do długości 8 znaków. Dopełnienie jest wypełniane napisem *+:

```
SELECT RPAD(name, 30, '.'), LPAD(price, 8, '*+')  
FROM products  
WHERE product_id < 4;
```

RPAD(NAME,30,'.')	LPAD(PRICE,8,'*+')
Nauka współczesna.....	*+*19,95
Chemia.....	*+*+*+30
Supernowa.....	*+*25,99

 Z tego przykładu wynika, że funkcje znakowe mogą operować na liczbach. Kolumna *price* zawiera liczbę, która została dopełniona po lewej stronie przez funkcję LPAD().

LTRIM(), RTRIM() i TRIM()

Funkcje LTRIM(), RTRIM() i TRIM() można opisać krótko:

- LTRIM(*x* [, *napis_przycinany*]) usuwa znaki z lewej strony *x*. Można przesyłać opcjonalny parametr *napis_przycinany* określający, które znaki mają zostać usunięte. Jeżeli parametr ten nie zostanie przesłany, będą domyślnie usuwane znaki spacji.
- RTRIM(*x* [, *napis_przycinany*]) usuwa znaki po prawej stronie *x*,
- TRIM([*napis_przycinany FROM*] *x*) usuwa znaki z lewej i prawej strony *x*. Można przesyłać opcjonalny parametr *napis_przycinany* określający, które znaki mają zostać usunięte. Jeżeli parametr ten nie zostanie przesłany, będą domyślnie usuwane znaki spacji.

Wszystkie trzy funkcje zostały wykorzystane w poniższym zapytaniu:

```
SELECT  
  LTRIM(' Cześć Edwardzie Nencki!'),  
  RTRIM('Cześć Ryszardzie Spacki!abcabc', 'abc'),  
  TRIM('0' FROM '000Cześć Mario Tupska!0000')  
FROM dual;
```

LTRIM('CZEŚĆEDWARDZIENENC RTRIM('CZEŚĆRYSZARDZIESPAC TRIM('0' FROM'000CZEŚĆ	Cześć Edwardzie Nencki! Cześć Ryszardzie Spacki! Cześć Mario Tupska!
---	--

NVL()

Funkcja NVL() konwertuje wartość NULL na inną. NVL(*x*, *wartość*) zwraca *wartość*, jeżeli *x* wynosi NULL. W przeciwnym razie zwraca *x*.

Poniższe zapytanie pobiera kolumny *customer_id* i *phone* z tabeli *customers*. Wartości NULL w kolumnie *phone* są przekształcane za pomocą funkcji NVL() na Nieznany numer telefonu:

```
SELECT customer_id, NVL(phone, 'Nieznany numer telefonu')
FROM customers;
```

```
CUSTOMER_ID NVL(PHONE, 'NIEZNANYNUME
```

```
-----  
1 800-555-1211  
2 800-555-1212  
3 800-555-1213  
4 800-555-1214  
5 Nieznany numer telefonu
```

Wartość z kolumny phone dla klienta nr 5 została przekształcona na Nieznany numer telefonu, ponieważ w tym wierszu kolumna phone ma wartość NULL.

NVL2()

Funkcja NVL2(*x*, *wartość1*, *wartość2*) zwraca *wartość1*, jeżeli *x* to nie NULL. W przeciwnym razie zwracana jest *wartość2*.

Poniższe zapytanie pobiera kolumny customer_id i phone z tabeli customers. Wartości inne niż NULL w kolumnie phone są konwertowane na napis Znany, a wartości NULL na napis Nieznany:

```
SELECT customer_id, NVL2(phone, 'Znany', 'Nieznany')
FROM customers;
```

```
CUSTOMER_ID NVL2(PHON
```

```
-----  
1 Znany  
2 Znany  
3 Znany  
4 Znany  
5 Nieznany
```

Wartości kolumny phone zostały przekształcone na Znane w przypadku klientów od 1. do 4., ponieważ w tych wierszach wartości kolumny są różne od NULL. W przypadku klienta nr 5 wartość jest konwertowana na Nieznany, ponieważ w tym wierszu w kolumnie phone występuje wartość NULL.

REPLACE()

Funkcja REPLACE(*x*, *szukany_napis*, *napis_zastępujący*) wyszukuje w *x* napis *szukany_napis* i zastępuje go napisem *napis_zastępujący*.

Poniższy przykład pobiera z tabeli products kolumnę name dla produktu nr 1 (którego nazwa to „Nauka współczesna”) i zastępuje za pomocą funkcji REPLACE() napis Nauka łańcuchem Fizyka:

```
SELECT REPLACE(name, 'Nauka', 'Fizyka')
FROM products
WHERE product_id = 1;

REPLACE(NAME,'NAUKA','FIZYKA')
```

```
Fizyka współczesna
```

 **Uwaga** Funkcja REPLACE() nie modyfikuje zawartości wiersza w bazie danych, a jedynie wiersz zwracany przez funkcję.

SOUNDEX()

Funkcja SOUNDEX(*x*) zwraca napis zawierający fonetyczną reprezentację *x*. To umożliwia porównywanie słów, które w języku angielskim brzmią podobnie, lecz mają inną pisownię.

SUBSTR()

Funkcja SUBSTR(*x*, *start* [, *długość*] zwraca podnapis napisu *x*, rozpoczynający się w pozycji określonej przez *start*. Pierwszy znak w *x* ma pozycję 1. Wartość *start* może być dodatnia lub ujemna. Wartość

dodatnia określa pozycję liczoną od początku x . Wartość ujemna określa pozycję liczoną od końca x . Można przesłać opcjonalny parametr *długość*, określający długość podnapisu.

Poniższe zapytanie wykorzystuje funkcję SUBSTR() do pobrania 7-znakowego podłańcucha rozpoczynającego się od pozycji 2 w kolumnie name tabeli products:

```
SELECT SUBSTR(name, 2, 7)
FROM products
WHERE product_id < 4;
```

```
SUBSTR(
-----
auka ws
hemia
upernow
```

Kolejne zapytanie wykorzystuje funkcję SUBSTR() do pobrania 3-znakowego podłańcucha rozpoczynającego się od pozycji -5 w kolumnie name tabeli products:

```
SELECT SUBSTR(name, -5, 3)
FROM products
WHERE product_id < 4;
```

```
SUB
---
zes
hem
rno
```

Używanie wyrażeń z funkcjami

W funkcjach możemy wykorzystywać nie tylko kolumny. Można przesłać dowolne poprawne wyrażenie, które zwraca串. Poniższe zapytanie wykorzystuje funkcję SUBSTR() do pobrania podnapisu mała z napisu Marysia miała małą owieczkę:

```
SELECT SUBSTR('Marysia miała małą owieczkę', 15, 4)
FROM dual;
```

```
SUBSTR
-----
mała
```

Łączenie funkcji

W instrukcji SQL można zastosować dowolną prawidłową kombinację funkcji. Poniższe zapytanie łączy funkcje UPPER() i SUBSTR(). Wyjście funkcji SUBSTR() jest przesyłane do funkcji UPPER():

```
SELECT name, UPPER(SUBSTR(name, 2, 8))
FROM products
WHERE product_id < 4;
```

NAME	UPPER(SUBSTR(NAME,2,8))
Nauka współczesna	AUKA WSP
Chemia	HEMIA
Supernowa	UPERNOWA



Możliwość łączenia funkcji nie jest ograniczona do funkcji znakowych — można łączyć z sobą funkcje różnego typu.

Funkcje numeryczne

Funkcje numeryczne służą do wykonywania obliczeń. Przyjmują one liczbę pochodzącą z kolumny lub dowolnego wyrażenia, którego wynikiem jest liczba. Następnie są wykonywane obliczenia i jest zwracana liczba. Przykładem funkcji numerycznej jest $\text{SQRT}(x)$, która zwraca pierwiastek kwadratowy x .

W tabeli 4.2 opisano niektóre funkcje numeryczne. W kolejnych podrozdziałach niektóre z tych funkcji zostaną opisane dokładniej.

Tabela 4.2. Funkcje numeryczne

Funkcja	Opis	Przykłady
$\text{ABS}(x)$	Zwraca wartość absolutną x	$\text{ABS}(10) = 10$ $\text{ABS}(-10) = 10$
$\text{ACOS}(x)$	Zwraca arcus cosinus x	$\text{ACOS}(1) = 0$ $\text{ACOS}(-1) = 3,14159265$
$\text{ASIN}(x)$	Zwraca arcus sinus x	$\text{ASIN}(1) = 1,57079633$ $\text{ASIN}(-1) = -1,57079633$
$\text{ATAN}(x)$	Zwraca arcus tangens x	$\text{ATAN}(1) = 0,785398163$ $\text{ATAN}(-1) = -0,78539816$
$\text{ATAN2}(x, y)$	Zwraca arcus tangens x i y	$\text{ATAN2}(1, -1) = 2,35619449$
$\text{BITAND}(x, y)$	Zwraca wynik bitowego AND dla x i y	$\text{BITAND}(0, 0) = 0$ $\text{BITAND}(0, 1) = 0$ $\text{BITAND}(1, 0) = 0$ $\text{BITAND}(1, 1) = 1$ $\text{BITAND}(1010, 1100) = 64$
$\text{COS}(x)$	Zwraca cosinus x , gdzie x jest kątem wyrażonym w radianach	$\text{COS}(90 * 3.1415926) = 1$ $\text{COS}(45 * 3.1415926) = -1$
$\text{COSH}(x)$	Zwraca cosinus hiperboliczny x	$\text{COSH}(3.1415926) = 11,5919527$
$\text{CEIL}(x)$	Zwraca najmniejszą liczbę całkowitą większą lub równą x	$\text{CEIL}(5.8) = 6$ $\text{CEIL}(-5.2) = -5$
$\text{EXP}(x)$	Zwraca wynik podniesienia liczby e do potęgi x , gdzie e w przybliżeniu wynosi 2,71828183	$\text{EXP}(1) = 2,71828183$ $\text{EXP}(2) = 7,3890561$
$\text{FLOOR}(x)$	Zwraca największą liczbę całkowitą mniejszą lub równą x	$\text{FLOOR}(5.8) = 5$ $\text{FLOOR}(-5.2) = 6$
$\text{GREATEST}(wartości)$	Zwraca największą z listy wartości.	$\text{GREATEST}(3, 4, 1) = 4$ $\text{GREATEST}(50 / 2, \text{EXP}(2)) = 25$
$\text{LEAST}(wartości)$	Zwraca najmniejszą z listy wartości	$\text{LEAST}(3, 4, 1) = 1$ $\text{LEAST}(50 / 2, \text{EXP}(2)) = 7,3890561$
$\text{LOG}(x, y)$	Zwraca logarytm o podstawie x liczby y	$\text{LOG}(2, 4) = 2$ $\text{LOG}(2, 5) = 2,32192809$
$\text{LN}(x)$	Zwraca logarytm naturalny liczby x	$\text{LN}(2.71828183) = 1$
$\text{MOD}(x, y)$	Zwraca resztę z dzielenia x przez y	$\text{MOD}(8, 3) = 2$ $\text{MOD}(8, 4) = 0$
$\text{POWER}(x, y)$	Zwraca wynik podniesienia liczby x do potęgi y	$\text{POWER}(2, 1) = 2$ $\text{POWER}(2, 3) = 8$
$\text{ROUND}(x [, y])$	Zwraca wynik zaokrąglenia liczby x do opcjonalnej liczby y miejsc po przecinku. Jeżeli y zostanie pominięta, x jest zaokrąglana do 0 miejsc po przecinku. Jeżeli y jest liczbą ujemną, x jest zaokrąglana po lewej stronie separatora dziesiętnego	$\text{ROUND}(5.75) = 6$ $\text{ROUND}(5.75, 1) = 5,8$ $\text{ROUND}(5.75, -1) = 10$
$\text{SIGN}(x)$	Zwraca -1 , jeżeli x jest liczbą ujemną, 1 , jeżeli jest liczbą dodatnią, lub 0 , jeśli x to zero	$\text{SIGN}(-5) = -1$ $\text{SIGN}(5) = 1$ $\text{SIGN}(0) = 0$
$\text{SIN}(x)$	Zwraca sinus liczby x	$\text{SIN}(0) = 0$
$\text{SINH}(x)$	Zwraca sinus hiperboliczny liczby x	$\text{SINH}(1) = 1,17520119$
$\text{SQRT}(x)$	Zwraca pierwiastek kwadratowy liczby x	$\text{SQRT}(25) = 5$ $\text{SQRT}(5) = 2,23606798$
$\text{TAN}(x)$	Zwraca tangens liczby x	$\text{TAN}(0) = 0$

Tabela 4.2. Funkcje numeryczne — ciąg dalszy

Funkcja	Opis	Przykłady
<code>TANH(x)</code>	Zwraca tangens hiperboliczny liczby x	$\text{TANH}(1) = 0,761594156$
<code>TRUNC(x [, y])</code>	Zwraca wynik obcięcia liczby x do opcjonalnych y miejsc dziesiętnych. Jeżeli y nie zostanie określona, x zostanie przycięta do zera miejsc dziesiętnych. Jeżeli y jest liczbą ujemną, x będzie przycinana po lewej stronie separatora dziesiętnego	$\text{TRUNC}(5.75) = 5$ $\text{TRUNC}(5.75, 1) = 5,7$ $\text{TRUNC}(5.75, -1) = 0$

ABS()

Funkcja `ABS(x)` oblicza wartość bezwzględna liczby x . Wartość bezwzględna liczby jest tą samą liczbą, ale bez żadnego znaku (dodatniego lub ujemnego). Poniższe zapytanie pobiera wartości bezwzględne liczb 10 i -10:

```
SELECT ABS(10), ABS(-10)
FROM dual;
```

ABS(10)	ABS(-10)
-----	-----
10	10

Parametry przesypane do funkcji numerycznych nie muszą być literałami liczbowymi. Dane wejściowe mogą również pochodzić z kolumny liczbowej w tabeli lub każdego poprawnego wyrażenia. Poniższe zapytanie pobiera wartości bezwzględne liczb obliczonych przez odjęcie 30 od wartości kolumny `price` tabeli `products` dla pierwszych trzech produktów:

```
SELECT product_id, price, price - 30, ABS(price - 30)
FROM products
WHERE product_id < 4;
```

PRODUCT_ID	PRICE	PRICE-30	ABS(PRICE-30)
-----	-----	-----	-----
1	19,95	-10,05	10,05
2	30	0	0
3	25,99	-4,01	4,01

CEIL()

Funkcja `CEIL(x)` (ang. *ceiling* — sufit) zwraca najmniejszą liczbę całkowitą równą x lub większą. Poniższe zapytanie za pomocą funkcji `CEIL()` zaokrąglą do góry wartości liczb 5,8 i -5,2:

```
SELECT CEIL(5.8), CEIL(-5.2)
FROM dual;
```

CEIL(5.8)	CEIL(-5.2)
-----	-----
6	-5

W wynikach otrzymujemy:

- Zaokrąglenie do góry liczby 5,8 wynosi 6, ponieważ 6 jest najmniejszą liczbą całkowitą większą od 5,8.
- Zaokrąglenie do góry liczby -5,2 wynosi -5, ponieważ -5,2 jest liczbą ujemną, a najmniejsza większa liczba całkowita od tej liczby to właśnie -5.

FLOOR()

Funkcja `FLOOR(x)` (ang. *floor* — podłoga) zwraca największą liczbę całkowitą równą x lub mniejszą. Poniższe zapytanie oblicza za pomocą funkcji `FLOOR()` część całkowitą liczb 5,8 i -5,2:

```
SELECT FLOOR(5.8), FLOOR(-5.2)
FROM dual;
```

```
FLOOR(5.8) FLOOR(-5.2)
-----
      5        -6
```

W wynikach otrzymujemy:

- Część całkowita liczby 5,8 wynosi 5, ponieważ jest to największa liczba całkowita mniejsza od 5,8.
- Część całkowita liczby -5,2 wynosi -6, ponieważ -5,2 jest liczbą ujemną i największa liczba całkowita mniejsza od tej wartości to właśnie -6.

GREATEST()

Funkcja GREATEST(*wartości*) zwraca największą wartość z listy *wartości*. Poniższe zapytanie za pomocą funkcji GREATEST() zwraca największą z wartości 3, 4 i 1:

```
SELECT GREATEST(3, 4, 1)
FROM dual;
```

```
GREATEST(3,4,1)
-----
      4
```

Następne zapytanie za pomocą funkcji GREATEST() zwraca największą z wartości powstałych w wyniku dzielenia 50 przez 2 i wykonania funkcji EXP(2):

```
SELECT GREATEST(50 / 2, EXP(2))
FROM dual;
```

```
GREATEST(50 / 2, EXP(2))
-----
      25
```

LEAST()

Funkcja LEAST(*wartości*) zwraca najmniejszą wartość z listy *wartości*. Poniższe zapytanie za pomocą funkcji LEAST() zwraca najmniejszą z wartości 3, 4 i 1.

```
SELECT LEAST(3, 4, 1)
FROM dual;
```

```
LEAST(3,4,1)
-----
      1
```

Następne zapytanie za pomocą funkcji LEAST() zwraca najmniejszą z wartości powstałych w wyniku dzielenia 50 przez 2 i wykonania funkcji EXP(2):

```
SELECT LEAST(50 / 2, EXP(2))
FROM dual;
```

```
LEAST(50 / 2, EXP(2))
-----
      7,3890561
```

POWER()

Funkcja POWER(*x*, *y*) zwraca wynik podniesienia liczby *x* do potęgi *y*. Poniższe zapytanie oblicza za pomocą funkcji POWER() wynik podniesienia liczby 2 do potęgi 1 i 3:

```
SELECT POWER(2, 1), POWER(2, 3)
FROM dual;
```

```
POWER(2,1) POWER(2,3)
-----
```

2	8
---	---

W wynikach widzimy, że:

- Podniesienie 2 do potęgi 1 jest równoważne liczbie 2, więc w wyniku otrzymujemy 2.
- Podniesienie liczby 2 do potęgi 3 jest równoważne działaniu $2 \cdot 2 \cdot 2$, więc w wyniku otrzymujemy 8.

ROUND()

Funkcja `ROUND(x, [y])` zwraca wynik zaokrąglenia liczby x do opcjonalnych y miejsc po przecinku. Jeżeli y nie zostanie określone, x zostanie zaokrąglone do zera miejsc po przecinku. Jeżeli y jest liczbą ujemną, x będzie zaokrąglane po lewej stronie separatora dziesiętnego.

Poniższe zapytanie wykorzystuje funkcję `ROUND()` do zaokrąglenia liczby 5,75 do 0, 1 i -1 miejsc po przecinku:

```
SELECT ROUND(5.75), ROUND(5.75, 1), ROUND(5.75, -1)
FROM dual;
```

```
ROUND(5.75) ROUND(5.75,1) ROUND(5.75,-1)
-----
```

6	5,8	10
---	-----	----

W wynikach widzimy, że:

- 5,75 zaokrąglona do zera miejsc po przecinku wynosi 6.
- 5,75 po zaokrągleniu do jednego miejsca po przecinku wynosi 5,8.
- 5,75 zaokrąglona do jednego miejsca dziesiętnego po lewej stronie separatora dziesiętnego (na co wskazuje znak ujemny) wynosi 10.

SIGN()

Funkcja `SIGN(x)` zwraca znak liczby x . Jeżeli x jest liczbą ujemną, funkcja zwraca -1, jeżeli x jest dodatnia, funkcja zwraca 1. Jeżeli x wynosi 0, funkcja zwraca 0. Poniższe zapytanie pobiera znaki liczb 5, -5 i 0:

```
SELECT SIGN(5), SIGN(-5), SIGN(0)
FROM dual;
```

```
SIGN(5) SIGN(-5) SIGN(0)
-----
```

1	-1	0
---	----	---

W wynikach widzimy, że:

- Znak liczby -5 to -1.
- Znak liczby 5 to 1.
- Znak liczby 0 to 0.

SQRT()

Funkcja `SQRT(x)` zwraca pierwiastek kwadratowy liczby x . Poniższe zapytanie oblicza pierwiastki kwadratowe liczby 25 i 5:

```
SELECT SQRT(25), SQRT(5)
FROM dual;
```

```
SQRT(25) SQRT(5)
-----
```

5	2,23606798
---	------------

TRUNC()

Funkcja `TRUNC(x [, y])` zwraca wynik obcięcia liczby x do opcjonalnych y miejsc dziesiętnych. Jeżeli y nie zostanie określony, x zostanie przycięta do zera miejsc dziesiętnych. Jeżeli y jest liczbą ujemną, x będzie przycinana po lewej stronie separatora dziesiętnego. Poniższe zapytanie przycina liczbę 5,75 do 0, 1 i -1 miejsca dziesiętnego:

```
SELECT TRUNC(5.75), TRUNC(5.75, 1), TRUNC(5.75, -1)
FROM dual;
```

TRUNC(5.75)	TRUNC(5.75,1)	TRUNC(5.75,-1)
5	5,7	0

W wynikach widzimy, że:

- 5,75 po przycięciu do zera miejsc dziesiętnych wynosi 5.
- 5,75 po przycięciu do jednego miejsca dziesiętnego po prawej stronie separatora dziesiętnego wynosi 5,7.
- 5,75 po przycięciu do jednego miejsca dziesiętnego po lewej stronie separatora dziesiętnego (na co wskazuje znak minus) wynosi 0.

Funkcje konwertujące

Czasami chcemy przekonwertować wartość z jednego typu danych na inny. Możemy chcieć zmienić format ceny produktu, która jest przechowywana jako liczba (na przykład 10 346,95), na napis zawierający symbol waluty i separator tysięcy (na przykład 10 346 zł). Do tego wykorzystujemy funkcje konwertujące, które konwertują wartość z jednego typu danych na inny.

W tabeli 4.3 opisano niektóre funkcje konwertujące.

Tabela 4.3. Funkcje konwertujące

Funkcja	Opis
<code>ASCIIISTR(x)</code>	Konwertuje x na napis ASCII, gdzie x może być napisem w dowolnym zestawie znaków. Znaki spoza zestawu ASCII są konwertowane do postaci \xxxx, gdzie xxxx reprezentuje wartość Unicode. Unicode umożliwia zapisywanie tekstu z wielu różnych języków.
<code>BIN_TO_NUM(x)</code>	Konwertuje liczbę binarną x na typ NUMBER
<code>CAST(x AS typ)</code>	Konwertuje x na kompatybilny typ z bazy danych, określony przez <i>typ</i>
<code>CHARTOROWID(x)</code>	Konwertuje x na ROWID. Jak już zostało zasygnalizowane w rozdziale 2., ROWID przechowuje lokalizację wiersza.
<code>COMPOSE(x)</code>	Konwertuje x na napis Unicode w tym samym zestawie znaków co x .
<code>CONVERT(x, źródłowy_zestaw_znaków, docelowy_zestaw_znaków)</code>	Konwertuje x z zestawu znaków <i>źródłowy_zestaw_znaków</i> na <i>docelowy_zestaw_znaków</i>
<code>DECODE(x, wyszukiwane, wynik, domyślna)</code>	Porównuje x z wartością <i>wyszukiwane</i> . Jeżeli są równe, funkcja zwraca <i>wynik</i> ; w przeciwnym razie zwraca wartość <i>domyślna</i> Więcej informacji na temat DECODE() znajduje się w rozdziale 7.
<code>DECOMPOSE(x)</code>	Konwertuje x na napis Unicode po dekompozycji napisu do tego samego zestawu znaków co x
<code>HEXTORAW(x)</code>	Konwertuje liczbę szesnastkową x na liczbę binarną (RAW). Funkcja zwraca liczbę binarną (RAW)

Tabela 4.3. Funkcje konwertujące — ciąg dalszy

Funkcja	Opis
NUMTODSINTERVAL(<i>x</i>)	Konwertuje liczbę <i>x</i> na INTERVAL DAY TO SECOND (funkcje związane z interwałami daty i czasu zostaną opisane w kolejnym rozdziale)
NUMTOYMINTERVAL(<i>x</i>)	Konwertuje liczbę <i>x</i> na INTERVAL YEAR TO MONTH
RAWTOHEX(<i>x</i>)	Konwertuje liczbę binarną (RAW) <i>x</i> na napis VARCHAR2, zawierający równoważną liczbę szesnastkową
RAWTONHEX(<i>x</i>)	Konwertuje liczbę binarną (RAW) <i>x</i> na napis NVARCHAR2, zawierający równoważną liczbę szesnastkową (NVARCHAR2 składa się z napisu, używając zestawu znaków narodowych)
ROWIDTOCHAR(<i>x</i>)	Konwertuje ROWID <i>x</i> na napis VARCHAR2
ROWIDTONCHAR(<i>x</i>)	Konwertuje ROWID <i>x</i> na napis NVARCHAR2
TO_BINARY_DOUBLE(<i>x</i>)	Konwertuje <i>x</i> na BINARY_DOUBLE (ta funkcja została wprowadzona w Oracle Database 10g)
TO_BINARY_FLOAT(<i>x</i>)	Konwertuje <i>x</i> na BINARY_FLOAT (ta funkcja została wprowadzona w Oracle Database 10g)
TO_BLOB(<i>x</i>)	Konwertuje <i>x</i> na duży obiekt binarny (BLOB). Typ BLOB jest używany do składowania dużych ilości danych binarnych. Więcej informacji na temat dużych obiektów znajduje się w rozdziale 15.
TO_CHAR(<i>x</i> [, <i>format</i>])	Konwertuje <i>x</i> na napis VARCHAR2. Można przesłać opcjonalny parametr <i>format</i> , określający sposób formatowania <i>x</i>
TO_CLOB(<i>x</i>)	Konwertuje <i>x</i> na duży obiekt znakowy (CLOB). Typ CLOB jest używany do przechowywania dużych ilości danych znakowych
TO_DATE(<i>x</i> [, <i>format</i>])	Konwertuje <i>x</i> na typ DATE. Więcej informacji na temat dat zamieszczono w rozdziale 5.
TO_DSINTERVAL(<i>x</i>)	Konwertuje napis <i>x</i> na INTERVAL DAY TO SECOND
TO_MULTI_BYTE(<i>x</i>)	Konwertuje jednobajtowe znaki w <i>x</i> na odpowiadające im znaki wielobajtowe. Typ zwracany jest taki sam jak typ <i>x</i>
TO_NCHAR(<i>x</i>)	Konwertuje <i>x</i> z zestawu znaków bazy danych na napis NVARCHAR2
TO_NCLOB(<i>x</i>)	Konwertuje <i>x</i> na duży obiekt NCLOB, używany do przechowywania dużych ilości danych znakowych ze znakami narodowymi
TO_NUMBER(<i>x</i> [, <i>format</i>])	Konwertuje <i>x</i> na typ NUMBER. Można przekazać też opcjonalny ciąg <i>format</i> .
TO_SINGLE_BYTE(<i>x</i>)	Konwertuje wielobajtowe znaki w <i>x</i> na odpowiadające im znaki jednobajtowe. Typ zwracany jest taki sam jak typ <i>x</i>
TO_TIMESTAMP(<i>x</i>)	Konwertuje napis <i>x</i> na typ TIMESTAMP. Więcej na temat znaczników czasu można znaleźć w rozdziale 5.
TO_TIMESTAMP_TZ(<i>x</i>)	Konwertuje napis <i>x</i> na typ TIMESTAMP WITH TIME ZONE
TO_YMINTERVAL(<i>x</i>)	Konwertuje napis <i>x</i> na typ INTERVAL YEAR TO MONTH. Więcej na temat odcinków czasu można znaleźć w rozdziale 5.
TRANSLATE(<i>x</i> , <i>napis_źródłowy</i> , <i>napis_docelowy</i>)	Konwertuje w <i>x</i> wszystkie wystąpienia <i>napis_źródłowy</i> na <i>napis_docelowy</i> . Więcej informacji na temat funkcji TRANSLATE() zamieszczono w rozdziale 7.
UNISTR(<i>x</i>)	Konwertuje znaki w <i>x</i> na znak NCHAR. NCHAR zapisuje znak, używając zestawu znaków narodowych.
	UNISTR() wspiera Unicode. Można wpisać bezpośrednio wartości Unicode znaków. Wartość Unicode ma postać \xxxx, gdzie xxxx to szesnastkowa liczba odpowiadająca znakowi w formacie Unicode.

Wiele z funkcji konwertujących będzie szczegółowo opisane w kolejnych podrozdziałach. Pozostałe funkcje z tabeli 4.3 zostaną omówione w dalszej części książki (te funkcje korzystają z typów danych opisanych w kolejnych rozdziałach i musisz najpierw dowiedzieć się, jak korzystać z tych typów danych).

Niektóre z funkcji konwertujących wykorzystują narodowe zestawy znaków i Unicode. Unicode umożliwia zapisywanie tekstu w wielu różnych językach. Więcej informacji o zestawach znaków narodowych i systemie Unicode można znaleźć w *Oracle Database Globalization Support Guide* opublikowanym przez Oracle Corporation.

ASCIISTR()

Funkcja ASCIISTR(*x*) konwertuje *x* na ciąg ASCII, gdzie *x* może być ciągiem znaków z dowolnego zestawu. Znaki spoza zestawu ASCII są konwertowane do postaci \xxxx, gdzie xxxx oznacza wartość Unicode.

Poniższe zapytanie wykorzystuje ASCIISTR() do konwersji ciągu 'ABC Ä CDE' do ciągu ASCII. Kropki nad A są nazywane umlautem. W przykładzie można zauważyc, że Ä zamieniane jest na \017D, co jest wartością Unicode dla litery Ä.

```
SELECT ASCIISTR('ABC Ä CDE')
  FROM DUAL;
```

```
ASCIISTR('ABC
-----
ABC \017D CDE
```

Następne zapytanie wykorzystuje ASCIISTR() do konwersji znaków zwróconych przez CHR() do wartości Unicode. Funkcja CHR(*x*) zwraca literę mającą wartość ASCII równą *x*.

```
SELECT ASCIISTR(CHR(128) || ' ' || CHR(129) || ' ' || CHR(130))
  FROM DUAL;
```

```
ASCIISTR(CHR(128)
-----
\20AC \0081 \201A
```

BIN_TO_NUM()

BIN_TO_NUM(*x*) zmienia liczbę dwójkową *x* na liczbę w systemie dziesiątkowym.

Poniższe zapytanie wykorzystuje BIN_TO_NUM(), by zamienić liczby dwójkowe na ich dziesiątkowe odpowiedniki:

```
SELECT
  BIN_TO_NUM(1, 0, 1),
  BIN_TO_NUM(1, 1, 0),
  BIN_TO_NUM(1, 1, 1, 0)
FROM dual;

BIN_TO_NUM(1,0,1) BIN_TO_NUM(1,1,0) BIN_TO_NUM(1,1,1,0)
-----
5           6           14
```

CAST()

Funkcja CAST(*x* AS *typ*) konwertuje *x* na kompatybilny typ z bazy danych, określany przez parametr *typ*. W tabeli 4.4 przedstawiono dopuszczalne konwersje typów (są oznaczone X).

Poniższe zapytanie przedstawia wykorzystanie funkcji CAST() do konwersji literalów na określone typy:

```
SELECT
  CAST(12345.67 AS VARCHAR2(10)),
  CAST('9A4F' AS RAW(2)),
  CAST('05-LIP-07' AS DATE),
  CAST(12345.678 AS NUMBER(10,2))
FROM dual;
```

Tabela 4.4. Dopuszczalne konwersje typów danych

Na typ	Z typu							
	BINARY_FLOAT BINARY_DOUBLE	CHAR VARCHAR2	NUMBER	DATE TIMESTAMP INTERVAL	RAW	ROWID UROWID	NCHAR NVARCHAR2	
BINARY_FLOAT	X	X	X				X	
BINARY_DOUBLE								
CHAR VARCHAR2	X	X	X	X	X	X		
NUMBER	X	X	X				X	
DATE TIMESTAMP INTERVAL		X		X				
RAW		X			X			
ROWID UROWID		X				X		
NCHAR NVARCHAR2	X		X	X	X	X	X	

```
CAST(12345 CAST CAST('05- CAST(12345.678ASNUMBER(10,2))
```

```
-----
```

```
12345,67 9A4F 05-LIP-07 12345,68
```

Można również konwertować wartości z kolumn tabeli na inny typ, co obrazuje poniższe zapytanie:

```
SELECT
  CAST(price AS VARCHAR2(10)),
  CAST(price + 2 AS NUMBER(7,2)),
  CAST(price AS BINARY_DOUBLE)
FROM products
WHERE product_id = 1;
```

```
CAST(PRICE CAST(PRICE+2ASNUMBER(7,2)) CAST(PRICEASBINARY_DOUBLE)
```

```
-----
```

```
19,95 21,95 1,995E+001
```

W rozdziale 5. poznasz kolejne przykłady prezentujące wykorzystanie funkcji `CAST()` do konwertowania dat, czasu i interwałów. Z rozdziału 14. dowiesz się, jak konwertować kolekcje za pomocą funkcji `CAST()`.

CHARTOROWID()

Funkcja `CHARTOROWID(x)` konwertuje `x` na `ROWID`. Jak zostało to już opisane w rozdziale 2., `ROWID` przechowuje lokalizację wiersza.

Poniższy przykład pobiera wartość identyfikatora wiersza dla klienta nr 1, a następnie przekazuje ją do funkcji `CHARTOROWID()`, aby pobrać dane klienta nr 1:

```
SELECT ROWID
FROM customers
WHERE customer_id = 1;
```

```
ROWID
```

```
-----
```

```
AAAW1zAAJAAAACsAAA
```

```
SELECT *
FROM customers
WHERE ROWID = CHARTOROWID('AAAW1zAAJAAAACsAAA');
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
```

```
-----
```

```
1 Jan Nikiel 01-STY-1965 800-555-1211
```



Jeśli wykonasz ten przykład, otrzymasz inną wartość identyfikatora wiersza.

COMPOSE()

Funkcja `COMPOSE(x)` zamienia *x* na ciąg Unicode. Zwrócony ciąg znaków korzysta z tego samego zestawu znaków co *x*.

Poniższe zapytanie korzysta z `COMPOSE()`, aby do litery *o* dodać umlaut (ö) i do litery *e* dodać półokrąg (ê):

```
SELECT
  COMPOSE('o' || UNISTR('\0308')),
  COMPOSE('e' || UNISTR('\0302'))
FROM DUAL;
```

C C
- -
ö ê

Funkcja `UNISTR(x)` konwertuje znaki z *x* do postaci ciągu znaków NCHAR, który przechowuje znaki w kodowaniu narodowym. Wartość Unicode ma postać \xxxx, gdzie xxxx jest szesnastkową reprezentacją znaku w formacie Unicode.



W zależności od ustawień Twojego systemu operacyjnego i bazy danych możesz uzyskać inne rezultaty wykonania przykładowych zapytań. Aby zobaczyć wszystkie znaki specjalne Unicode, można użyć programu SQL Developer do wykonania zapytania. Dotyczy to również innych przykładów z tego rozdziału, w których pojawiają się znaki specjalne.

CONVERT()

Funkcja `CONVERT(x, źródłowy_zestaw_znaków, docelowy_zestaw_znaków)` zmienia zestaw znaków, w jakim kodowany jest ciąg *x*.

Poniższe zapytanie wykorzystuje funkcję `CONVERT()` do konwersji ciągu znaków z kodowania US7ASCII (7-bitowe ASCII) na kodowanie WE8ISO8859P1 (8-bitowe ISO 8859-1 dla Europy Zachodniej):

```
SELECT CONVERT('Ä È Í Ö Å B C', 'US7ASCII', 'WE8ISO8859P1')
FROM DUAL;
```

CONVERT('ÄÈÍÖÅBC'

A E I O A B C

DECOMPOSE()

Funkcja `DECOMPOSE(x)` konwertuje *x* na ciąg znaków Unicode po rozłączeniu w tym samym kodowaniu co *x*. Na przykład *o* z umlautem (ö) jest zwracane w postaci znaku *o* i umlautu.

Poniższe zapytanie pokazuje użycie `DECOMPOSE()` do oddzielenia umlautu z ö i półokrągu z ê:

```
SELECT DECOMPOSE('öê')
FROM DUAL;
```

DECO

o^e^

HEXTORAW()

Funkcja `HEXTORAW(x)` zamienia *x* zawierający liczbę szesnastkową na postać binarną (RAW). Funkcja zwraca liczbę.

Poniższe zapytanie wykorzystuje `HEXTORAW()` do konwersji liczby szesnastkowej:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('41414743'))
FROM DUAL;
```

```
UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('41414743'))
```

```
-----  
AAGC
```

Funkcja UTL_RAW.CAST_TO_VARCHAR2() zamienia czystą wartość liczbową zwróconą przez HEXTORAW() na typ VARCHAR2 możliwy do wyświetlenia w SQL*Plus.

RAWTOHEX()

Funkcja RAWTOHEX(*x*) zamienia czystą wartość binarną *x* na ciąg znaków VARCHAR2 zawierający reprezentację szesnastkową tej wartości.

Poniższe zapytanie wykorzystuje RAWTOHEX() do konwersji wartości na liczbę szesnastkową:

```
SELECT RAWTOHEX(41414743)  
FROM DUAL;
```

```
RAWTOHEX(4  
-----  
C42A2A302C
```

ROWIDTOCHAR()

Funkcja ROWIDTOCHAR(*x*) zamienia identyfikator wiersza *x* na ciąg znaków typu VARCHAR2.

Poniższe zapytanie pokazuje wykorzystanie funkcji ROWIDTOCHAR() w klauzuli WHERE:

```
SELECT ROWID, customer_id, first_name, last_name  
FROM customers  
WHERE ROWIDTOCHAR(ROWID) LIKE '%sAAA%';
```

```
ROWID          CUSTOMER_ID FIRST_NAME LAST_NAME  
-----  
AAAW1zAAJAAAAsAAA      1 Jan        Nikiel
```



W Twojej bazie danych identyfikator wiersza będzie inny.

TO_BINARY_DOUBLE()

Funkcja TO_BINARY_DOUBLE(*x*) konwertuje *x* na wartość typu BINARY_DOUBLE.

Poniższy przykład pokazuje wykorzystanie funkcji TO_BINARY_DOUBLE() do konwersji liczby:

```
SELECT TO_BINARY_DOUBLE(1087623)  
FROM dual;
```

```
TO_BINARY_DOUBLE(1087623)  
-----  
1.088E+006
```

TO_BINARY_FLOAT()

Funkcja TO_BINARY_FLOAT(*x*) konwertuje *x* na wartość typu BINARY_FLOAT.

Poniższy przykład pokazuje wykorzystanie funkcji TO_BINARY_FLOAT() do konwersji liczby:

```
SELECT TO_BINARY_FLOAT(10623)  
FROM dual;
```

```
TO_BINARY_FLOAT(10623)  
-----  
1.062E+004
```

TO_CHAR()

Funkcja TO_CHAR(*x* [, *format*]) konwertuje *x* na napis. Można przesyłać opcjonalny parametr *format*, określający sposób formatowania *x*. Struktura parametru *format* zależy od tego, czy *x* jest liczbą, czy datą. Z tego podrozdziału dowiesz się, jak za pomocą funkcji TO_CHAR() konwertować liczby na napisy, a w kolejnym rozdziale opisano, jak konwertować daty na napisy.

Przyjrzyjmy się kilku prostym zapytaniom, konwertującym liczbę na napis za pomocą funkcji `TO_CHAR()`. Poniższe zapytanie konwertuje na napis liczbę 12345,67:

```
SELECT TO_CHAR(12345.67)
FROM dual;
```

```
TO_CHAR(
-----
12345,67
```

Kolejne zapytanie konwertuje liczbę 12345,67 na napis zgodnie z formatem określonym przez `99G999D99`. Przy polskich ustawieniach narodowych zwracany jest łańcuch zawierający znak spacji jako separator tysięcy i przecinek jako separator dziesiąteń:

```
SELECT TO_CHAR(12345.67, '99G999D99')
FROM dual;
```

```
TO_CHAR(12
-----
12 345,67
```

Opcjonalny napis *format*, który można przesłać do funkcji `TO_CHAR()`, posiada wiele parametrów mających wpływ na napis zwracany przez funkcję. Niektóre z tych parametrów zostały opisane w tabeli 4.5.

Przyjrzyjmy się kolejnym przykładom konwertowania liczb na napisy za pomocą funkcji `TO_CHAR()`. Tabela 4.6 przedstawia przykłady wywołań funkcji `TO_CHAR()` oraz zwrócone wyniki.

Jeżeli spróbujemy sformatować liczbę, która zawiera zbyt wiele cyfr dla przeslanego formatu, funkcja `TO_CHAR()` zwróci ciąg znaków `#`, na przykład:

```
SELECT TO_CHAR(12345678.90, '99,999.99')
FROM dual;
```

```
TO_CHAR(12
-----
#####
```

Funkcja `TO_CHAR()` zwróciła znaki `#`, ponieważ liczba 12345678,90 zawiera więcej cyfr niż limit dopuszczony przez format `99,999.99`.

Za pomocą funkcji `TO_CHAR()` można również konwertować na napisy kolumny zawierające liczby. Na przykład poniższe zapytanie wykorzystuje funkcję `TO_CHAR()` do przeprowadzenia konwersji wartości z kolumny `price` tabeli `products` na napisy:

```
SELECT product_id, 'Cena produktu wynosi' || TO_CHAR(price, '99D99L')
FROM products
WHERE product_id < 5;
```

```
PRODUCT_ID 'CENAPRODUKTUWYNOSI'||TO_CHAR(PRICE,
-----
1 Cena produktu wynosi      19,95zŁ
2 Cena produktu wynosi      30,00zŁ
3 Cena produktu wynosi      25,99zŁ
4 Cena produktu wynosi      13,95zŁ
```

TO_MULTI_BYTE()

Funkcja `TO_MULTI_BYTE(x)` zamienia jednobajtowe znaki z *x* na ich odpowiedniki wielobajtowe. Zwracane dane są tego samego typu co *x*.

Poniższe zapytanie wykorzystuje funkcję `TO_MULTI_BYTE()` do konwersji litery na jej reprezentację wielobajtową:

```
SELECT TO_MULTI_BYTE('A')
FROM dual;
```

```
T
-
A
```

Tabela 4.5. Parametry formatujące liczby

Parametr	Przykład formatu	Opis
9	999	Zwraca cyfry na określonych pozycjach wraz z początkowym znakiem minus, jeżeli liczba jest ujemna
0	0999	0999 zwraca liczbę poprzedzaną zerami
	9990	9990 zwraca liczbę kończącą zerami
.	999.99	Zwraca kropkę jako separator dziesiętny na określonej pozycji
,	999,99	Zwraca przecinek na określonej pozycji (w przypadku polskich ustawień narodowych w takim przypadku separatorem dziesiętnym musi być kropka)
\$	\$999	Poprzedza liczbę znakiem dolara
B	B9.99	Jeżeli całkowita część liczby stałoprzecinkowej jest zerem, zwraca znak spacji zamiast zera
C	999C	Zwraca symbol ISO waluty na określonej pozycji. Symbol pochodzi z parametru NLS_ISO_CURRENCY bazy danych i jest definiowany przez administratora bazy danych
D	9D99	Zwraca symbol separatora dziesiętnego na określonej pozycji. Symbol pochodzi z parametru NLS_NUMERIC_CHARACTER bazy danych (przy polskich ustawieniach narodowych jest to domyślnie przecinek)
EEEE	9.99EEEE	Zwraca liczbę, używając notacji naukowej
FM	FM90.9	Usuwa początkowe i końcowe spacje z liczby
G	9G999	Zwraca symbol separatora grupy na określonej pozycji. Symbol pochodzi z parametru NLS_NUMERIC_CHARACTER bazy danych
L	999L	Zwraca lokalny symbol waluty na określonej pozycji. Symbol pochodzi z parametru NLS_CURRENCY bazy danych
MI	999MI	Zwraca liczbę ujemną ze znakiem minus umieszczonym na końcu. Na końcu liczby dodatniej jest umieszczana spacja
PR	999PR	Zwraca liczbę ujemną w nawiasach ostrokatnych (< >) oraz liczbę dodatnią poprzedzoną i zakończoną znakiem spacji
RN	RN	Zwraca liczbę w zapisie rzymskim. RN zwraca numerały zapisywane wielkimi literami,
Rn	rn	a rn zwraca numerały zapisywane małymi literami. Liczba musi być liczbą całkowitą z przedziału od 1 do 3999
S	S999 999S	S999 zwraca liczbę ujemną poprzedzoną znakiem minus, a liczbę dodatnią poprzedzoną znakiem plus 999S zwraca liczbę ujemną zakończoną znakiem minus, a liczbę dodatnią zakończoną znakiem plus
TM	TM	Zwraca liczbę z użyciem jak najmniejszej liczby znaków. Domyślnie obowiązuje format TM9, który zwraca liczby, używając zapisu stałoprzecinkowego, chyba że liczba znaków jest większa od 64. W takim przypadku liczba jest zwracana w notacji naukowej
U	U999	Zwraca drugi symbol waluty (na przykład euro) na określonej pozycji. Symbol pochodzi z parametru NLS_DUAL_CURRENCY bazy danych
V	99V99	Zwraca liczbę pomnożoną razy 10 ^x , gdzie x jest liczbą znaków 9 za znakiem V. Jeżeli jest to konieczne, liczba jest zaokrąglana
X	XXXX	Zwraca liczbę w formacie szesnastkowym. Jeżeli nie jest ona całkowita, jest zaokrąglana do liczby całkowitej

TO_NUMBER()

Funkcja `TO_NUMBER(x [, format])` konwertuje x na liczbę. Można przesłać opcjonalny napis *format*, określający format x. W napisie *format* mogą znajdować się takie same parametry jak te wymienione w tabeli 4.4.

Tabela 4.6. Przykłady zastosowania funkcji TO_CHAR

Wywołanie funkcji TO_CHAR()	Wynik
TO_CHAR(12345.67, '99999.99')	12345.67
TO_CHAR(12345.67, '99,999.99')	12,345.67
TO_CHAR(-12345.67, '99,999.99')	-12,345.67
TO_CHAR(12345.67, '099,999.99')	012,345.67
TO_CHAR(12345.67, '99,999.9900')	12,345.6700
TO_CHAR(12345.67, '\$99,999.99')	\$12,345.67
TO_CHAR(0.67, 'B9.99')	.67
TO_CHAR(12345.67, 'C99,999.99')	PLN12345,67
TO_CHAR(12345.67, '99999D99')	12345,67
TO_CHAR(12345.67, '99999.99EEEE')	1.23E+04
TO_CHAR(0012345.6700, 'FM99999.99')	12345.67
TO_CHAR(12345.67, '99999G99')	123 46
TO_CHAR(12345.67, 'L99,999.99')	zł12345.67
TO_CHAR(-12345.67, '99,999.99MI')	12345.67-
TO_CHAR(-12345.67, '99,999.99PR')	<12345.67>
TO_CHAR(2007, 'RN')	MMVII
TO_CHAR(12345.67, 'TM')	12345,67
TO_CHAR(12345.67, 'U99,999.99')	zł12,345.67
TO_CHAR(12345.67, '99999V99')	1234567

Poniższe zapytanie konwertuje na liczbę napis 970,13, korzystając z funkcji TO_NUMBER():

```
SELECT TO_NUMBER('970,13')
FROM dual;
```

```
TO_NUMBER('970,13')
-----
970,13
```

Kolejne zapytanie konwertuje napis 970,13 na liczbę za pomocą funkcji TO_NUMBER(), a następnie dodaje do tej liczby 25,5:

```
SELECT TO_NUMBER('970,13') + 25.5
FROM dual;
```

```
TO_NUMBER('970,13')+25.5
-----
995,63
```

Kolejne zapytanie konwertuje napis -1 234,56 zł na liczbę za pomocą funkcji TO_NUMBER, przesyłając do niej napis formatujący 9G999D99L:

```
SELECT TO_NUMBER('-1 234,56Z', '9G999D99L')
FROM dual;
```

```
TO_NUMBER('-1234,56Z','9G999D99L')
-----
-1234,56
```

TO_SINGLE_BYTE()

Funkcja TO_SINGLE_BYTE(*x*) konwertuje wielobajtowe znaki z *x* na ich jednobajtowe odpowiedniki. Zwracane dane są tego samego typu co *x*.

Poniższe zapytanie wykorzystuje funkcję TO_SINGLE_BYTE() do konwersji litery na jej jednobajtową reprezentację:

```
SELECT TO_SINGLE_BYTE('A')
FROM DUAL;
```

```
T
-
A
```

UNISTR()

Funkcja UNISTR(*x*) konwertuje znaki z *x* na napis NCHAR. NCHAR przechowuje znaki, wykorzystując wybrane kodowanie znaków. UNISTR() wspiera również Unicode. Można w ciągu znaków podać kod Unicode znaku. Kod Unicode ma postać \xxxx, gdzie xxxx to szesnastkowa wartość opisująca znak w kodzie Unicode.

Poniższe zapytanie wykorzystuje UNISTR(), by wyświetlić umlaut i półokrąg:

```
SELECT UNISTR('\0308'), UNISTR('\0302')
FROM DUAL;
```

```
U U
- -
.. ^
```

Funkcje wyrażeń regularnych

W tym podrozdziale zostały opisane wyrażenia regularne i związane z nimi funkcje bazy danych Oracle, które umożliwiają wyszukiwanie wzorców znaków w napisie. Założymy, że dysponujemy poniższą listą lat:

```
1964
1965
1966
1967
1968
1969
1970
1971
```

i chcemy z niej pobrać te z przedziału od 1965 do 1968. Możemy to zrobić za pomocą wyrażenia regularnego:

```
^196[5-8]$
```

Wyrażenie regularne zawiera zbiór **metaznaków**. W tym przykładzie są nim ^, [5-8] i \$:

- ^ oznacza początek napisu,
- [5-8] — przedział znaków od 5 do 8,
- \$ — pozycję w napisie.

^196 oznacza więc napis rozpoczynający się od 196, a [5-8]\$ — napis kończący się cyfrą 5, 6, 7 lub 8, dlatego warunek ^196[5-8]\$ jest spełniany przez 1965, 1966, 1967 i 1968, czyli dokładnie przez te lata, które chcieliśmy pobrać z listy.

W następnym przykładzie został wykorzystany ten napis będący cytatem z *Romea i Julii*:

Lecz cicho! Co za blask strzelił tam z okna!

Założymy, że chcemy wyszukać podnapis blask. Posłuży do tego poniższe wyrażenie regularne:

```
b[:alpha:]{4}
```

W tym wyrażeniu regularnym metaznakami są [:alpha:] i {4}:

- [:alpha:] oznacza znak alfanumeryczny od A do Z i od a do z,
- {4} powtarza czterokrotnie wcześniejsze dopasowanie.

Po połączeniu b, [:alpha:] i {4} uzyskujemy wyrażenie spełniane przez sekwencję pięciu liter, rozpoczynającą się literą b, dlatego też wyrażenie regularne b[:alpha:]{4} jest spełniane przez blask z napisu.

W tabeli 4.7 opisano niektóre metaznaki możliwe do wykorzystania w wyrażeniach regularnych, a także ich znaczenie i przykłady zastosowania.

Tabela 4.7. Metaznaki w wyrażeniach regularnych

Metaznaki	Znaczenie	Przykłady
\	Spełniany przez znak specjalny lub literał albo wykonuje odwołanie wsteczne	\n oznacza znak nowego wiersza \ \ oznacza \ \(oznacza (\) oznacza)
^	Oznacza początek napisu	^A jest spełniane przez A, jeżeli ta litera jest pierwszym znakiem napisu
\$	Oznacza koniec napisu	\$B jest spełniane przez B, jeżeli ta litera jest ostatnim znakiem napisu
*	Oznacza zero lub więcej wystąpień poprzedzającego znaku	ba*rk jest spełniane przez brk, bark, baark itd.
+	Oznacza co najmniej jedno wystąpienie poprzedzającego znaku	ba+rk jest spełniane przez bark, baark itd., ale nie przez brk
?	Oznacza zero lub jedno wystąpienie poprzedzającego znaku	ba?rk jest spełniane tylko przez brk i bark
{n}	Oznacza dokładnie n wystąpień znaku. n musi być liczbą całkowitą	hob{2} it jest spełniane przez hobbit
{n,m}	Oznacza przynajmniej n i maksymalnie m wystąpień znaku, gdzie n i m są liczbami całkowitymi	hob{2,3} it jest spełniane tylko przez hobbit i hobbit
.	Oznacza jeden dowolny znak oprócz NULL	hob.it jest spełniane przez hobait, hobbit itd.
(wzorzec)	Podwyrażenie spełniane przez określony wzorzec. Za pomocą podwyrażeń można tworzyć złożone wyrażenia regularne. Można uzyskać dostęp do poszczególnych wystąpień, zwanych napisami przechwyconymi	telefo(n nia) jest spełnianie przez telefon i telefonia
x y	Jest spełniane przez x lub y, gdzie x i y stanowią co najmniej znak	wojna pokój jest spełniane przez słowo wojna lub pokój
[abc]	Jest spełniane przez każdy wymieniony znak	[ab]bc jest spełniane zarówno przez abc, jak i bbc
[a-z]	Jest spełniane przez każdy znak z określonego zakresu	[a-c]bc jest spełniane przez abc, bbc i cbc
[: :]	Okręsła klasę znaku i jest spełniane przez dowolny znak z tej klasy	[:alnum:] jest spełniane przez znaki alfanumeryczne 0 – 9, A – Z i a – z [:alpha:] jest spełniane przez litery A – Z i a – z [:blank:] jest spełniane przez znak spacji lub tabulacji [:digit:] jest spełniane przez cyfry 0 – 9 [:graph:] jest spełniane przez znak drukowalny [:lower:] jest spełniane przez małe litery alfabetu a – z [:print:] jest podobne do [:graph:], ale uwzględnia spację [:punct:] jest spełniane przez znaki interpunkcyjne ., " itd.

Tabela 4.7. Metaznaki w wyrażeniach regularnych — ciąg dalszy

Metaznaki	Znaczenie	Przykłady
[: :]	Okręsła klasę znaku i jest spełniane przez dowolny znak z tej klasy	[:space:] jest spełniane przez znaki odstępu [:upper:] jest spełniane przez wielkie litery alfabetu A – Z [:xdigit:] jest spełniane przez wszystkie znaki dopuszczalne w liczbie szesnastkowej: 0 – 9, A – F, a – f
[..]	Jest spełniane przez jeden symbol łączony, na przykład w symbolu wieloznakowym	Brak przykładu
[==]	Okręsła klasy równoważności	Brak przykładu
\n	Jest to odwołanie wsteczne do wcześniej przechwyconego elementu; <i>n</i> musi być dodatnią liczbą całkowitą	(.) \1 jest spełniane przez dwa identyczne znaki następujące po sobie. (.) przechwytyuje każdy znak oprócz NULL, a \1 powtarza przechwytcenie, tym samym przechwytyując jeszcze raz ten sam znak. Dlatego też wyrażenie jest spełniane przez dwa identyczne znaki następujące po sobie

W Oracle Database 10g Release 2 wprowadzono kilka metaznaków używanych w Perlu. Zostały one opisane w tabeli 4.8.

Tabela 4.8. Metaznaki dodane z języka Perl

Metaznaki	Opis
\d	cyfra
\D	znak niebędący cyfrą
\w	słowo
\W	niesłowo
\s	znak białej spacji
\S	znak inny niż biała spacja
\A	spełniane tylko przez początek napisu lub jego koniec, jeżeli znajduje się przed znakiem nowego wiersza
\Z	spełniane tylko przez koniec napisu
*?	spełniane przez 0 lub więcej wystąpień wcześniejszego elementu wzorca
+?	spełniane przez co najmniej jedno wystąpienie wcześniejszego elementu wzorca
??	spełniane przez 0 lub jedno wystąpienie wcześniejszego elementu wzorca
{n}	spełniane przez dokładnie <i>n</i> wystąpień wcześniejszego elementu wzorca
{n,}	spełniane przez przynajmniej <i>n</i> wystąpień wcześniejszego elementu wzorca
{n,m}	spełniane przez przynajmniej <i>n</i> , ale mniej niż <i>m</i> wystąpień wcześniejszego elementu wzorca

W tabeli 4.9 opisano funkcje operujące na wyrażeniach regularnych. Zostały one wprowadzone w Oracle Database 10g i rozszerzone w wersji 11g, co zostało zaznaczone w tabeli.

W kolejnych podrozdziałach zostaną dokładniej opisane funkcje operujące na wyrażeniach regularnych.

REGEXP_LIKE()

Funkcja REGEXP_LIKE(*x, wzorzec [, opcja_dopasowania]*) przeszukuje *x* zgodnie z wyrażeniem regularnym zdefiniowanym przez parametr *wzorzec*. Można również przesyłać opcjonalny parametr *opcja_dopasowania*, który może być jednym z poniższych znaków:

Tabela 4.9. Funkcje operujące na wyrażeniach regularnych

Funkcja	Opis
REGEXP_LIKE(<i>x</i> , <i>wzorzec</i> [, <i>opcja_dopasowania</i>])	Przeszukuje <i>x</i> zgodnie z wyrażeniem regularnym zdefiniowanym przez parametr <i>wzorzec</i> . Można również przesłać opcjonalny parametr <i>opcja_dopasowania</i> , który może mieć jedną z poniższych wartości: <ul style="list-style-type: none"> ■ 'c' określa, że podczas wyszukiwania wielkość liter będzie miała znaczenie (jest to opcja domyślna) ■ 'I' określa, że podczas wyszukiwania wielkość liter nie będzie miała znaczenia ■ 'n' umożliwia użycie operatora spełnianego przez dowolny znak ■ 'm' powoduje traktowanie <i>x</i> jako wielu linii
REGEXP_INSTR(<i>x</i> , <i>wzorzec</i> [, <i>start</i> [, <i>wystąpienie</i> [, <i>opcja_zwracania</i> [, <i>opcja_dopasowania</i> [, <i>opcja_podwyrażenia</i>]]]])	Przeszukuje <i>x</i> zgodnie z wyrażeniem regularnym <i>wzorzec</i> i zwraca pozycję, na której występuje <i>wzorzec</i> . Można przesłać opcjonalne parametry: <ul style="list-style-type: none"> ■ <i>start</i> określa pozycję, od której zostanie rozpoczęte przeszukiwanie. Domyślną wartością jest 1, czyli pierwszy znak w <i>x</i> ■ <i>wystąpienie</i> określa, które wystąpienie <i>wzorzec</i> powinno zostać zwrócone. Domyślną wartością jest 1, co oznacza, że funkcja zwróci pozycję pierwszego wystąpienia <i>wzorzec</i> ■ <i>opcja_zwracania</i> określa, jaka liczba całkowita zostanie zwrócona. 0 określa, że zwrócona liczba całkowita będzie oznaczała pozycję pierwszego znaku w <i>x</i>. 1 oznacza, że zwrócona liczba całkowita będzie oznaczała pozycję znaku w <i>x</i> po wystąpieniu <i>wzorzec</i> ■ <i>opcja_dopasowania</i> zmienia domyślny sposób dopasowywania do wzorca. Opcje są takie same jak w przypadku funkcji REGEXP_LIKE() ■ <i>opcja_podwyrażenia</i> (nowość w Oracle Database 11g) ma następujące działanie: w przypadku wzorca z podwyrażeniami <i>opcja_podwyrażenia</i> jest nieujemną liczbą całkowitą od 0 do 9, określającą, które podwyrażenie we <i>wzorzec</i> jest celem funkcji. Na przykład wyrażenie: 0123((abc)(de)fghi)45(678) zawiera pięć podwyrażeń: abcdefghi, abcdef, abc, de oraz 678. Jeżeli <i>opcja_podwyrażenia</i> będzie równa 0, zostanie zwrócona pozycja całego wyrażenia <i>wzorzec</i>. Jeżeli <i>wzorzec</i> nie zawiera prawidłowej liczby podwyrażeń, funkcja zwróci 0. Jeżeli <i>opcja_podwyrażenia</i> ma wartość NULL, funkcja zwróci NULL. Domyślną wartością <i>opcja_podwyrażenia</i> jest 0
REGEXP_REPLACE(<i>x</i> , <i>wzorzec</i> [, <i>napis_zastępujący</i> [, <i>start</i> [, <i>wystąpienie</i> [, <i>opcja_dopasowania</i>]]]])	Wyszukuje <i>wzorzec</i> w <i>x</i> i zastępuje go napisem <i>napis_zastępujący</i> . Znaczenie pozostałych opcji zostało opisane powyżej
REGEXP_SUBSTR(<i>x</i> , <i>wzorzec</i> [, <i>start</i> [, <i>wystąpienie</i> [, <i>opcja_dopasowania</i> [, <i>opcja_podwyrażenia</i>]]]])	Zwraca podnapis <i>x</i> zgodny z <i>wzorzec</i> . Wyszukiwanie rozpoczyna się od pozycji określonej przez <i>start</i> . Znaczenie pozostałych opcji zostało opisane powyżej. Znaczenie <i>opcja_podwyrażenia</i> (nowej w Oracle Database 11g) jest takie samo jak w przypadku funkcji REGEXP_INSTR()
REGEXP_COUNT(<i>x</i> , <i>wzorzec</i> [, <i>start</i> [, <i>opcja_dopasowania</i>]])	Wprowadzone w Oracle Database 11g. Wyszukuje <i>wzorzec</i> w <i>x</i> i zwraca liczbę wystąpień <i>wzorzec</i> . Można przesłać poniższe opcjonalne parametry: <ul style="list-style-type: none"> ■ <i>start</i> określa pozycję, od której rozpoczęcie się wyszukiwanie. Domyślną wartością jest 1, co oznacza pierwszy znak w napisie <i>x</i> ■ <i>opcja_dopasowania</i> zmienia domyślny sposób dopasowywania. Ma takie samo znaczenie jak w przypadku funkcji REGEXP_LIKE()

- 'c' określającym, że podczas wyszukiwania wielkość liter będzie miała znaczenie (jest to ustalenie domyślne),
- 'I' określającym, że podczas wyszukiwania wielkość liter nie będzie miała znaczenia,
- 'n' umożliwiającym użycie operatora spełnianego przez dowolny znak,
- 'm' traktującym x jak wiele wierszy.

Poniższe zapytanie pobiera za pomocą funkcji REGEXP_LIKE() informacje o klientach, których data urodzenia zawiera się w przedziale od 1965 do 1968:

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(TO_CHAR(dob, 'YYYY'), '^196[5-8]$');
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	Jan	Nikiel	65/01/01
2	Lidia	Stal	68/02/05

Kolejne zapytanie pobiera informacje o klientach, których imię rozpoczyna się literą j lub J. Należy zwrócić uwagę, że do funkcji REGEXP_LIKE() jest przesyłane wyrażenie regularne ^j, a opcja dopasowania jest ustaliona na i (i oznacza, że w wyszukiwaniu nie będzie brana pod uwagę wielkość liter, więc w tym przykładzie ^j jest spełniane przez j i J):

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(first_name, '^j', 'i');
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	Jan	Nikiel	65/01/01
5	Jadwiga	Mosiądz	70/05/20

REGEXP_INSTR()

Funkcja REGEXP_INSTR(x, wzorzec [, start [, wystąpienie [, opcja_zwracania [, opcja_dopasowania [, opcja_podwyrażenia]]]]]) wyszukuje wzorzec w x. Funkcja zwraca pozycję, na której wzorzec występuje w x (pozycje rozpoczynają się od 1).

Poniższe zapytanie zwraca pozycję spełniającą wyrażenie regularne b[[alpha:]]{4}, korzystając z funkcji REGEXP_INSTR():

```
SELECT REGEXP_INSTR('Lecz cicho! Co za blask strzelił tam z okna!', 
    'b[[alpha:]]{4}') AS wynik
FROM dual;
```

WYNIK
19

Została zwrócona liczba 19, która określa pozycję litery b ze słowa blask w całym napisie.

Następne zapytanie zwraca pozycję drugiego wystąpienia spełniającego wzorzec r[[alpha:]]{2}, rozpoczynając od pozycji 1:

```
SELECT REGEXP_INSTR('Idzie rak, nieborak.', 'r[[alpha:]]{2}', 1,2) AS wynik
FROM dual;
```

WYNIK
17

Kolejne zapytanie zwraca pozycję drugiego wystąpienia litery o, rozpoczynając wyszukiwanie od pozycji 10:

```
SELECT REGEXP_INSTR('Lecz cicho! Co za blask strzelił tam z okna!', 'o', 10, 2) AS wynik
FROM dual;
```

WYNIK

14

REGEXP_REPLACE()

Funkcja REGEXP_REPLACE(*x*, *wzorzec* [, *napis_zastępujący* [, *start* [, *wystąpienie* [, *opcja_dopasowania*]]]]) wyszukuje *wzorzec* w *x* i zastępuje go napisem *napis_zastępujący*.

Poniższe zapytanie za pomocą funkcji REGEXP_REPLACE() zastępuje podnapis zgodny z wyrażeniem regularnym o[[:alpha:]]{3} napisem szafy:

```
SELECT REGEXP_REPLACE('Lecz cicho! Co za blask strzelił tam z okna!',
  'o[[:alpha:]]{3}', 'szafy') AS wynik
FROM dual;
```

WYNIK

Lecz cicho! Co za blask strzelił tam z szafy!

Słowo okna zostało zastąpione słowem szafy.

REGEXP_SUBSTR()

Funkcja REGEXP_SUBSTR(*x*, *wzorzec* [, *start* [, *wystąpienie* [, *opcja_dopasowania* [, *opcja_podwyrażenia*]]]]) wyszukuje w *x* podnapis zgodny z *wzorzec*. Przeszukiwanie jest rozpoczynane od pozycji określonej przez *start*.

Poniższe zapytanie zwraca podnapis zgodny z wyrażeniem regularnym b[[:alpha:]]{3}, korzystając z funkcji REGEXP_SUBSTR():

```
SELECT REGEXP_SUBSTR('Lecz cicho! Co za blask strzelił tam z okna!',
  'b[[:alpha:]]{4}') AS wynik
FROM dual;
```

WYNIK

blask

REGEXP_COUNT()

Funkcja REGEXP_COUNT() została wprowadzona w Oracle Database 11g. Funkcja REGEXP_COUNT(*x*, *wzorzec* [, *start* [, *opcja_dopasowania*]]) wyszukuje *wzorzec* w *x* i zwraca liczbę jego wystąpień. Można przesłać opcjonalny parametr *start*, określający znak w *x*, od którego rozpoczęcie się wyszukiwanie, oraz opcjonalny parametr *opcja_dopasowania*, definiujący opcje dopasowania.

Poniższe zapytanie za pomocą funkcji REGEXP_COUNT() zwraca liczbę wystąpień wyrażenia regularnego r[[:alpha:]]{2} w napisie:

```
SELECT REGEXP_COUNT('Idzie rak, nieborak', 'r[[:alpha:]]{2}') AS wynik
FROM dual;
```

WYNIK

2

Została zwrócona liczba 2, co oznacza, że w napisie wystąpiły dwa dopasowania do wyrażenia regularnego.

Funkcje agregujące

Funkcje prezentowane dotychczas operują na pojedynczych wierszach i zwracają jeden wiersz wyników dla każdego wiersza wejściowego. W tym podrozdziale poznamy funkcje agregujące, które operują na grupie wierszy i zwracają jeden wiersz wyników.



Funkcje agregujące są czasem nazywane grupującymi, ponieważ operują na grupach wierszy.

W tabeli 4.10 opisano niektóre funkcje agregujące, z których wszystkie zwracają typ NUMBER. Oto kilka właściwości funkcji agregujących, o których warto pamiętać podczas używania ich:

- Funkcje agregujące mogą być używane z dowolnymi, prawidłowymi wyrażeniami. Na przykład funkcje COUNT(), MAX() i MIN() mogą być używane z liczbami, napisami i datami.
- Wartość NULL jest ignorowana przez funkcje agregujące, ponieważ wskazuje, że wartość jest nieznana i z tego powodu nie może zostać użyta w funkcji.
- Wraz z funkcją agregującą można użyć słowa kluczowego DISTINCT, aby wykluczyć z obliczeń powtarzające się wpisy.

Tabela 4.10. Funkcje agregujące

Funkcja	Opis
AVG(x)	Zwraca średnią wartość x
COUNT(x)	Zwraca liczbę wierszy zawierających x , zwróconych przez zapytanie
MAX(x)	Zwraca maksymalną wartość x
MEDIAN(x)	Zwraca medianę x
MIN(x)	Zwraca minimalną wartość x
STDDEV(x)	Zwraca odchylenie standardowe x
SUM(x)	Zwraca sumę x
VARIANCE(x)	Zwraca wariancję x

Funkcje agregujące przedstawione w tabeli 4.10 zostaną szerzej opisane w kolejnych podrozdziałach.

AVG()

Funkcja AVG(x) oblicza średnią wartość x . Poniższe zapytanie zwraca średnią cenę produktów. Należy zwrócić uwagę, że do funkcji AVG() jest przesyłana kolumna price z tabeli products:

```
SELECT AVG(price)
FROM products;
```

```
AVG(PRICE)
-----
19,7308333
```

Funkcje agregujące mogą być używane z dowolnymi prawidłowymi wyrażeniami. Na przykład poniższe zapytanie przesyła do funkcji AVG() wyrażenie price + 2. Na skutek tego do wartości price w każdym wierszu jest dodawane 2, a następnie jest obliczana średnia wyników:

```
SELECT AVG(price + 2)
FROM products;
```

```
AVG(PRICE+2)
-----
21,7308333
```

W celu wyłączenia z obliczeń identycznych wartości można użyć słowa kluczowego DISTINCT. Na przykład w poniższym zapytaniu użyto go do wyłączenia identycznych wartości z kolumny price podczas obliczania średniej za pomocą funkcji AVG():

```
SELECT AVG(DISTINCT price)
FROM products;
```

```
AVG(DISTINCTPRICE)
-----
20,2981818
```

Należy zauważyć, że w tym przypadku średnia jest nieco wyższa niż wartość zwrócona przez pierwsze zapytanie prezentowane w tym podrozdziale. Jest tak dlatego, ponieważ wartość kolumny price dla produktu nr 2 (13,49) jest taka sama jak dla produktu nr 7. Jest uznawana za duplikat i wyłączana z obliczeń wykonywanych przez funkcję AVG(), dlatego średnia w tym przykładzie jest nieco wyższa.

COUNT()

Funkcja COUNT(x) oblicza liczbę wierszy zwróconych przez zapytanie. Poniższe zapytanie zwraca liczbę wierszy w tabeli products, korzystając z funkcji COUNT():

```
SELECT COUNT(product_id)
FROM products;
```

```
COUNT(PRODUCT_ID)
```

```
-----
```

```
12
```

 **Wskazówka** Należy unikać stosowania gwiazdki (*) jako argumentu funkcji COUNT(), ponieważ obliczenie wyniku może zająć więcej czasu. Zamiast tego należy przesłać nazwę kolumny z tabeli lub użyć pseudokolumny ROWID. (Jak wiesz z rozdziału 2., kolumna ROWID zawiera wewnętrzną lokalizację wiersza w bazie danych Oracle).

Poniższe zapytanie przesyła ROWID do funkcji COUNT() i zwraca liczbę wierszy w tabeli products:

```
SELECT COUNT(ROWID)
FROM products;
```

```
COUNT(ROWID)
```

```
-----
```

```
12
```

MAX() i MIN()

Funkcje MAX(x) i MIN(x) zwracają maksymalną i minimalną wartość x. Poniższe zapytanie zwraca maksymalną i minimalną wartość z kolumny price tabeli products, korzystając z funkcji MAX() i MIN():

```
SELECT MAX(price), MIN(price)
FROM products;
```

```
MAX(PRICE) MIN(PRICE)
```

```
-----
```

```
49,99 10,99
```

Funkcje MAX() i MIN() mogą być używane ze wszystkimi typami danych, włącznie z napisami i datami. Gdy używamy MAX() z napisami, są one porządkowane alfabetycznie, z „maksymalnym” napisem umieszczonym na dole listy i „minimalnym” napisem umieszczonym na górze listy. Na przykład na takiej liście napis Albert znajdzie się przed napisem Zenon.

Poniższy przykład pobiera „maksymalny” i „minimalny” napis z kolumny name tabeli products, korzystając z funkcji MAX() i MIN():

```
SELECT MAX(name), MIN(name)
FROM products;
```

```
MAX(NAME) MIN(NAME)
```

```
-----
```

```
Z innej planety 2412: Powrót
```

W przypadku dat, „maksymalną” datą jest najpóźniejszy moment, „minimalną” — najwcześniejszy. Poniższe zapytanie pobiera maksymalną i minimalną wartość z kolumny dob tabeli customers, korzystając z funkcji MAX() i MIN():

```
SELECT MAX(dob), MIN(dob)
FROM customers;
```

```
MAX(DOB)  MIN(DOB)
-----
16-MAR-71 01-STY-65
```

STDDEV()

Funkcja `STDDEV(x)` oblicza odchylenie standardowe x . Jest ono funkcją statystyczną i jest definiowane jako pierwiastek kwadratowy wariancji (pojęcie wariancji zostanie opisane za chwilę).

Poniższe zapytanie oblicza odchylenie standardowe wartości w kolumnie `price` tabeli `products`, korzystając z funkcji `STDDEV()`:

```
SELECT STDDEV(price)
FROM products;
```

```
STDDEV(PRICE)
-----
11,0896303
```

SUM()

Funkcja `SUM(x)` dodaje wszystkie wartości w x i zwraca wynik. Poniższe zapytanie zwraca sumę wartości z kolumny `price` tabeli `products`, korzystając z funkcji `SUM()`:

```
SELECT SUM(price)
FROM products;
```

```
SUM(PRICE)
-----
236,77
```

VARIANCE()

Funkcja `VARIANCE(x)` oblicza wariancję x . Wariancja jest funkcją statystyczną i jest definiowana jako rozpiętość czy zróżnicowanie grupy liczb w próbce. Jest równa kwadratowi odchylenia standardowego.

Poniższe zapytanie oblicza wariancję wartości w kolumnie `price` tabeli `products`, korzystając z funkcji `VARIANCE()`:

```
SELECT VARIANCE(price)
FROM products;
```

```
VARIANCE(PRICE)
-----
122,979899
```

Grupowanie wierszy

Czasami chcemy pogrupować wiersze tabeli i uzyskać jakieś informacje na temat tych grup wierszy. Na przykład możemy chcieć uzyskać średnie ceny różnych typów produktów z tabeli `products`.

Grupowanie wierszy za pomocą klauzuli GROUP BY

Klauzula `GROUP BY` grupuje wiersze w bloki ze wspólną wartością jakiejś kolumny. Na przykład poniższe zapytanie grupuje wiersze z tabeli `products` w bloki z tą samą wartością `product_type_id`:

```
SELECT product_type_id
FROM products
GROUP BY product_type_id;
```

```
PRODUCT_TYPE_ID
-----
1
```

```

2
4
3

```

Należy zauważyć, że w zestawie wyników znajduje się tylko jeden wiersz dla każdego bloku wierszy z tą samą wartością `product_type_id`, a także, że między 1. i 2. występuje luka (wkrótce dowiemy się, dla czego się tam znajduje).

W tabeli `products` znajdują się dwa wiersze, dla których `product_type_id` jest równe 1, cztery wiersze, dla których `product_type_id` jest równe 2 itd. Te wiersze są grupowane w osobne bloki za pomocą klauzuli `GROUP BY` — każdy blok zawiera wszystkie wiersze z tą samą wartością `product_type_id`. Pierwszy zawsze zawiera dwa wiersze, drugi zawsze cztery wiersze itd.

Luka między wierszami 1. i 2. jest spowodowana tym, że w tabeli `products` występuje wiersz, w którym `product_type_id` ma wartość `NULL`. Ten wiersz jest przedstawiony w poniższym przykładzie:

```

SELECT product_id, name, price
FROM products
WHERE product_type_id IS NULL;

```

PRODUCT_ID	NAME	PRICE
12	Pierwsza linia	13,49

Ponieważ wartość `product_type_id` w tym wierszu wynosi `NULL`, klauzula `GROUP BY` w poprzednim zapytaniu grupuje te wiersze w osobnym bloku. Wiersz w zestawie wyników jest pusty, ponieważ wartość `product_type_id` dla tego bloku wynosi `NULL` — stąd luka między wierszami 1. i 2.

Używanie wielu kolumn w grupie

W klauzuli `GROUP BY` można określić kilka kolumn. Na przykład poniższe zapytanie zawiera w klauzuli `GROUP BY` kolumny `product_id` i `customer_id` z tabeli `purchases`:

```

SELECT product_id, customer_id
FROM purchases
GROUP BY product_id, customer_id;

```

PRODUCT_ID	CUSTOMER_ID
1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
3	3

Używanie funkcji agregujących z grupami wierszy

Do funkcji agregującej można przesyłać bloki wierszy. Wykoną ona obliczenia na grupie wierszy z każdego bloku i zwróci jedną wartość dla każdego bloku. Na przykład aby uzyskać liczbę wierszy z tą samą wartością `product_type_id` w tabeli `products`, musimy:

- pogrupować wiersze w bloki z tą samą wartością `product_type_id` za pomocą klauzuli `GROUP BY`,
- zliczyć wiersze w każdym bloku za pomocą funkcji `COUNT(ROWID)`.

Demonstruje to poniższe zapytanie:

```

SELECT product_type_id, COUNT(ROWID)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;

```

```
PRODUCT_TYPE_ID COUNT(ROWID)
-----
1             2
2             4
3             2
4             3
               1
```

Należy zauważyć, że w zestawie wyników znajduje się pięć wierszy, z których każdy odpowiada jednemu lub kilku wierszom z tabeli products, które zostały pogrupowane według wartości product_type_id. W zestawie wyników widzimy, że w dwóch wierszach product_type_id ma wartość 1, cztery wiersze mają wartość product_type_id równą 2 itd. Ostatni wiersz zestawu wyników wskazuje, że występuje jeden wiersz, w którym product_type_id ma wartość NULL (jest to wspomniany wcześniej wiersz „Pierwsza linia”).

Przejdzmy do innego przykładu. Aby uzyskać średnią cenę różnych typów produktów z tabeli products, musimy:

- za pomocą klauzuli GROUP BY pogrupować wiersze w bloki z tą samą wartością product_type_id,
- za pomocą funkcji AVG(price) obliczyć średnią cenę w każdym bloku wierszy.

Demonstruje to poniższe zapytanie:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

```
PRODUCT_TYPE_ID AVG(PRICE)
-----
1             24,975
2             26,22
3             13,24
4             13,99
               13,49
```

Każda grupa wierszy z tą samą wartością product_type_id jest przesyłana do funkcji AVG(). Następnie funkcja ta oblicza średnią cenę w każdej grupie. Jak widzimy w zestawie wyników, średnia cena w grupie produktów z product_type_id równym 1 wynosi 24,975, a średnia cena w grupie produktów z product_type_id równym 2 wynosi 26,22. Należy zauważyć, że w ostatnim wierszu zestawu wyników jest wyświetlana średnia cena równa 13,49. Jest to po prostu cena produktu „Pierwsza linia”, czyli jedynego wiersza, w którym product_type_id ma wartość NULL.

Z klauzulą GROUP BY możemy używać dowolnych funkcji agregujących. Na przykład kolejne zapytanie pobiera wariancję cen produktów dla każdego product_type_id:

```
SELECT product_type_id, VARIANCE(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

```
PRODUCT_TYPE_ID VARIANCE(PRICE)
-----
1             50,50125
2             280,8772
3                 ,125
4                  7
               0
```

Warto pamiętać, że nie musimy umieszczać kolumn wykorzystywanych w klauzuli GRUP BY bezpośrednio za instrukcją SELECT. Na przykład poniższe zapytanie ma takie samo znaczenie jak poprzednie, ale product_type_id zostało pominięte w klauzuli SELECT:

```
SELECT VARIANCE(price)
FROM products
GROUP BY product_type_id
```

```
ORDER BY product_type_id;
```

```
VARIANCE(PRICE)
```

```
-----  
50,50125  
280,8772  
,125  
7  
0
```

Wywołanie funkcji agregującej można również umieścić w klauzuli ORDER BY, co pokazuje poniższe zapytanie:

```
SELECT VARIANCE(price)  
FROM products  
GROUP BY product_type_id  
ORDER BY VARIANCE(price);
```

```
VARIANCE(PRICE)
```

```
-----  
0  
,125  
7  
50,50125  
280,8772
```

Nieprawidłowe użycie funkcji agregujących

Jeżeli zapytanie zawiera funkcję agregującą i pobiera kolumny nieujęte w niej, należy je umieścić w klauzuli GROUP BY. Jeśli o tym zapomnimy, zostanie wyświetlony komunikat o błędzie: ORA-00937: to nie jest jednogrupowa funkcja grupowa.

Na przykład poniższe zapytanie próbuje pobrać dane z kolumny product_type_id oraz obliczyć AVG(price), pominięto w nim jednak klauzulę GROUP BY dla product_type_id:

```
SQL> SELECT product_type_id, AVG(price)  
2 FROM products;  
SELECT product_type_id, AVG(price)  
*
```

BŁĄD w linii 1:
ORA-00937: to nie jest jednogrupowa funkcja grupowa

Błąd występuje, ponieważ baza danych nie wie, co zrobić z kolumną product_type_id. Zastanówmy się nad tym: zapytanie próbuje użyć funkcji agregującej AVG(), która operuje na wielu wierszach, ale próbuje również pobrać wartości product_type_id dla pojedynczych wierszy. Nie można zrobić tego jednocześnie. Należy zastosować klauzulę GROUP BY, aby wiersze z tą samą wartością product_type_id zostały zgrupowane. Wówczas baza danych prześle te grupy wierszy do funkcji AVG().



Jeżeli zapytanie zawiera funkcję agregującą i pobiera kolumny, które nie zostały w niej ujęte, należy je umieścić w klauzuli GROUP BY.

Poza tym nie można używać funkcji agregujących do ograniczania wierszy za pomocą klauzuli WHERE. W przeciwnym razie zostanie wyświetlony komunikat o błędzie: ORA-00934: funkcja grupowa nie jest tutaj dozwolona:

```
SQL> SELECT product_type_id, AVG(price)  
2 FROM products  
3 WHERE AVG(price) > 20  
4 GROUP BY product_type_id;  
WHERE AVG(price) > 20  
*
```

BŁĄD w linii 3:
ORA-00934: funkcja grupowa nie jest tutaj dozwolona

Błąd występuje, ponieważ klauzula WHERE służy jedynie do filtrowania *pojedynczych* wierszy, a nie grup, do czego służy klauzula HAVING, opisana poniżej.

Filtrowanie grup wierszy za pomocą klauzuli HAVING

Klauzula HAVING służy do filtrowania grup wierszy. Umieszcza się ją za klauzulą GROUP BY:

```
SELECT ...
FROM ...
WHERE
GROUP BY ...
HAVING ...
ORDER BY ...;
```

 **Uwaga** Klauzula GROUP BY może być używana bez klauzuli HAVING, ale klauzula HAVING musi być używana z klauzulą GROUP BY.

Załóżmy, że chcemy przejrzeć typy produktów, których średnia cena jest większa niż 20 zł. W tym celu musimy:

- za pomocą klauzuli GROUP BY pogrupować wiersze w bloki o tej samej wartości product_type_id,
- za pomocą klauzuli HAVING ograniczyć zwrócone wyniki jedynie do tych, w których średnia cena jest większa od 20 zł.

Demonstruje to poniższe zapytanie:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) > 20;

PRODUCT_TYPE_ID AVG(PRICE)
-----
1      24,975
2      26,22
```

Jak widzimy, zostały wyświetlane jedynie wiersze, w których średnia cena jest większa niż 20 zł.

Jednoczesne używanie klauzul WHERE i GROUP BY

Klauzule WHERE i GROUP BY mogą być użyte w tym samym zapytaniu. Wówczas klauzula WHERE najpierw filtryuje zwracane wiersze, a następnie klauzula GROUP BY grupuje pozostałe w bloki.

Na przykład w poniższym zapytaniu:

- Klauzula WHERE filtryuje wiersze tabeli products, wybierając jedynie te, w których wartość price jest mniejsza od 15.
- Klauzula GROUP BY grupuje pozostałe wiersze według wartości kolumny product_type_id.

```
SELECT product_type_id, AVG(price)
FROM products
WHERE price < 15
GROUP BY product_type_id
ORDER BY product_type_id;

PRODUCT_TYPE_ID AVG(PRICE)
-----
2      14,45
3      13,24
4      12,99
5      13,49
```

Jednoczesne używanie klauzul WHERE, GROUP BY i HAVING

Klauzule WHERE, GROUP BY i HAVING mogą zostać użyte w tym samym zapytaniu. Wówczas klauzula WHERE najpierw filtryuje zwracane wiersze, a następnie klauzula GROUP BY grupuje pozostałe wiersze w bloki, po czym klauzula HAVING filtryuje grupy wierszy.

Na przykład w poniższym zapytaniu:

- Klauzula WHERE filtryuje wiersze tabeli products, wybierając jedynie te, w których wartość price jest mniejsza od 15.
- Klauzula GROUP BY grupuje pozostałe wiersze według wartości kolumny product_type_id.
- Klauzula HAVING filtryuje grupy wierszy, wybierając jedynie te, w których średnia cena jest wyższa niż 13.

```
SELECT product_type_id, AVG(price)
FROM products
WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY product_type_id;

PRODUCT_TYPE_ID AVG(PRICE)
-----
2      14,45
3      13,24
      13,49
```

Porównajmy te wyniki z poprzednim przykładem: po filtracji została usunięta grupa wierszy, w których product_type_id ma wartość 4, a to dlatego, że w tej grupie wierszy średnia cena jest mniejsza od 13.

Ostatnie zapytanie wykorzystuje klauzulę ORDER BY AVG(price) w celu uporządkowania wyników według średniej ceny:

```
SELECT product_type_id, AVG(price)
FROM products WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY AVG(price);

PRODUCT_TYPE_ID AVG(PRICE)
-----
3      13,24
      13,49
2      14,45
```

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- w bazie danych Oracle występują dwie główne grupy funkcji: jednowierszowe i agregujące,
- funkcje jednowierszowe operują na pojedynczych wierszach i zwracają jeden wiersz wyników dla każdego wiersza wejściowego,
- występuje pięć głównych typów funkcji jednowierszowych: znakowe, numeryczne, konwertujące, dat i wyrażenia regularnych,
- funkcje agregujące operują na wielu wierszach i zwracają jeden wiersz wyników,
- bloki wierszy mogą być grupowane za pomocą klauzuli GROUP BY,
- grupy wierszy mogą być filtrowane za pomocą klauzuli HAVING.

W następnym rozdziale zawarto szczegółowe informacje o datach i czasie.

ROZDZIAŁ

5

Składowanie oraz przetwarzanie dat i czasu

Z tego rozdziału dowiesz się, jak:

- Przetwarzać i składować konkretne daty i czas, nazywane wyrażeniami typu **DataGodzina** (ang. *datetime*). Przechowują one rok, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minuty, sekundy, części ułamkowe sekundy i informację o strefie czasowej.
- Używać **znaczników czasu** (datowników, ang. *timestamps*) do składowania określonych dat i czasu. Znacznik czasu przechowuje rok, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minuty, sekundy, ułamki sekund oraz strefę czasową.
- Używać **interwałów** do składowania długości odcinka czasu. Przykładem interwału jest rok i trzy miesiące.

Proste przykłady składowania i pobierania dat

Informacje o dacie i czasie są zapisywane w bazie danych w polach typu **DATE**. Domyślny format wyświetlenia dat (przy polskich ustawieniach narodowych) to YY/MM/DD, gdzie:

Segment	Opis	Przykład
DD	dwie cyfry dnia	05
MM	dwie cyfry miesiąca	02
YY	dwie ostatnie cyfry roku	68

Poniższa instrukcja **INSERT** wstawia wiersz do tabeli **customers**, zapisując w kolumnie **dob** datę 68/02/05:

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    6, 'Fyderyk', 'Żelazo', '68/02/05', '800-555-1215'
);
```

Aby wprowadzić literal daty do bazy danych, można również użyć słowa kluczowego **DATE**. Data musi być zapisana zgodnie ze standardem ANSI YYYY-MM-DD, gdzie:

Segment	Opis	Przykład
YYYY	cztery cyfry roku	1972
MM	dwie cyfry miesiąca	10
DD	dwie cyfry dnia	25

Na przykład aby wprowadzić datę 25 października 1972 roku, użyjemy DATE '1972-10-25'. Poniższa instrukcja INSERT wstawia do tabeli customers wiersz, określając wartość dla kolumny dob jako DATE '1972-10-25':

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    7, 'Stefan', 'Tytan', DATE '1972-10-25', '800-555-1215'
);
```

Domyślnie baza danych zwraca daty w formacie YY/MM/DD, gdzie YY oznacza dwie ostatnie cyfry roku. W poniższym przykładzie jest najpierw wykonywane zapytanie pobierające wiersze z tabeli customers:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
6	Fydryk	Żelazo	68/02/05	800-555-1215
7	Stefan	Tytan	72/10/25	800-555-1215

W powyższym zestawie wyników kolumna dob dla klienta nr 4 jest pusta, ponieważ występuje w niej wartość NULL.

Poniższa instrukcja ROLLBACK pozwoli cofnąć zmiany wprowadzone przez poprzednie instrukcje INSERT.

```
ROLLBACK;
```



Jeżeli faktycznie uruchomiono dwie instrukcje INSERT, należy cofnąć zmiany za pomocą instrukcji ROLLBACK. Dzięki temu baza zostanie zachowana w stanie początkowym, a wyniki zwracane przez kolejne zapytania będą zgodne z prezentowanymi w książce.

Konwertowanie typów DataGodzina za pomocą funkcji TO_CHAR() i TO_DATE()

Baza danych Oracle posiada funkcje umożliwiające konwertowanie wartości danych jednego typu na inny. Z tego podrozdziału dowiesz się, jak za pomocą funkcji TO_CHAR() i TO_DATE() konwertować daty i godziny na napisy i odwrotnie. W tabeli 5.1 opisano funkcje TO_CHAR() i TO_DATE().

Tabela 5.1. Funkcje konwertujące TO_CHAR() i TO_DATE()

Funkcja	Opis
TO_CHAR(<i>x</i> [, <i>format</i>])	Konwertuje <i>x</i> na napis. Można również przesłać opcjonalny parametr <i>format</i> dla <i>x</i> . Z poprzedniego rozdziału wiesz już, jak można za pomocą tej funkcji konwertować liczby na napisy. Z tego rozdziału dowiesz się, jak konwertować daty i czas na napisy
TO_DATE(<i>x</i> [, <i>format</i>])	Konwertuje napis <i>x</i> na typ DATE

Kolejny podrozdział pokazuje, jak wykorzystać funkcję TO_CHAR() do konwersji daty i czasu na napis. Później dowiesz się, jak wykorzystać TO_DATE() do konwersji napisu na typ DATE.

Konwersja daty i czasu na napis za pomocą funkcji TO_CHAR()

Za pomocą funkcji TO_CHAR(*x* [, *format*]) można przekonwertować datę i godzinę *x* na napis. Można również przesłać opcjonalny parametr *format* dla *x*. Przykładowym formatem może być MONTH DD, YYYY, gdzie:

Segment	Opis	Przykład
MONTH	pełna nazwa miesiąca zapisana wielkimi literami	STYCZEŃ
DD	dwie cyfry dnia	02
YYYY	cztery cyfry roku	1965

Poniższe zapytanie wykorzystuje funkcję TO_CHAR() do konwersji danych z kolumny dob tabeli customers na łańcuch o formacie DD MONTH, YYYY:

```
SELECT customer_id, TO_CHAR(dob, 'DD MONTH, YYYY')
FROM customers;
```

```
CUSTOMER_ID TO_CHAR(DOB,'DDMONTH,YYYY')
```

```
-----  
1 01 STYCZEŃ , 1965  
2 05 LUTY , 1968  
3 16 MARZEC , 1971  
4  
5 20 MAJ , 1970
```

Kolejne zapytanie pobiera bieżącą datę i godzinę z bazy danych, korzystając z funkcji SYSDATE(), a następnie konwertuje te dane na łańcuch za pomocą funkcji TO_CHAR() i formatu DD MONTH, YYYY, HH24:MI:SS. Fragment definiujący godziny określa format 24-godzinny oraz powoduje umieszczenie w łańcuchu również minut i sekund:

```
SELECT TO_CHAR(SYSDATE, 'DD MONTH, YYYY, HH24:MI:SS')
FROM dual;
```

```
TO_CHAR(SYSDATE,'DDMONTH,YYYY,HH24:MI:SS')
```

```
-----  
05 LISTOPAD , 2011, 12:34:36
```

Gdy konwertujemy datę i godzinę na napis za pomocą funkcji TO_CHAR(), możemy określić szereg parametrów formatujących, mających wpływ na wygląd zwracanego napisu. Niektóre z nich zostały opisane w tabeli 5.2.

Tabela 5.2. Parametry formatujące daty i godziny

Aspekt	Parametr	Opis	Przykład
Wiek	CC	Dwie cyfry określające wiek	21
	SCC	Dwie cyfry oraz znak minus (-) w przypadku lat p.n.e.	-10
Kwartał	Q	Kwartał określony jedną cyfrą	1
Rok	YYYY	Wszystkie cztery cyfry roku	2008
	IYYY	Wszystkie cztery cyfry roku w formacie ISO (ISO to International Organization for Standardization)	2008
	RRRR	Wszystkie cztery cyfry zaokrąglonego roku (zależne od bieżącego roku). Więcej informacji na ten temat znajduje się w podrozdziale „Jak Oracle interpretuje lata dwucyfrowe?”	2008
	SYYYY	Wszystkie cztery cyfry roku oraz znak minus (-) w przypadku lat p.n.e.w przypadku lat p.n.e.	-1001
	Y,YYY	Wszystkie cztery cyfry roku z separatorem po pierwszej cyfrze	2 008
	YYY	Ostatnie trzy cyfry roku	008
	IYY	Ostatnie trzy cyfry roku w formacie ISO	008
	YY	Ostatnie dwie cyfry roku	08

Tabela 5.2. Parametry formatujące daty i godziny — ciąg dalszy

Aspekt	Parametr	Opis	Przykład
	IY	Ostatnie dwie cyfry roku w formacie ISO	06
	RR	Ostatnie dwie cyfry zaokrąglonego roku, który zależy od sposobu reprezentacji roku bieżącego. Więcej informacji na ten temat znajduje się w podrozdziale „Jak Oracle interpretuje lata dwucyfrowe?”	08
	Y	Ostatnia cyfra roku	8
	I	Ostatnia cyfra roku w formacie ISO	8
	YEAR	Rok zapisany słownie, wielkimi literami, w języku angielskim	TWO THOUSAND EIGHT
	Year	Rok zapisany słownie w języku angielskim, z wielkimi pierwszymi literami	Two Thousand Eight
Miesiąc	MM	Dwie cyfry określające miesiąc	01
	MONTH	Pełna nazwa miesiąca zapisana wielkimi literami, dopełniana po prawej stronie znakami spacji aż do uzyskania napisu dziewięciognakowego	STYCZEŃ
	Month	Pełna nazwa miesiąca rozpoczynana wielką literą, dopełniana po prawej stronie znakami spacji aż do uzyskania napisu dziewięciognakowego	Styczeń
	MON	Pierwsze trzy litery nazwy miesiąca, zapisane wielkimi literami	STY
	Mon	Pierwsze trzy litery nazwy miesiąca — pierwsza litera wielka	Sty
	RM	Numer miesiąca w zapisie rzymskim	Czwarty miesiąc (kwiecień) będzie reprezentowany jako IV
Tydzien	WW	Dwie cyfry oznaczające tydzień roku	02
	IW	Dwie cyfry oznaczające tydzień roku w formacie ISO	02
	W	Jedna cyfra określająca tydzień w miesiącu	2
Dzień	DDD	Trzycyfrowa liczba określająca dzień roku	103
	DD	Dwucyfrowa liczba określająca dzień miesiąca	31
	D	Jedna cyfra określająca dzień tygodnia	5
	DAY	Nazwa dnia tygodnia zapisana wielkimi literami	SOBOTA
	Day	Nazwa dnia tygodnia zapisana wielką literą	Sobota
	DY	Pierwsze trzy litery nazwy dnia zapisane wielkimi literami	SOB
	Dy	Pierwsze trzy litery nazwy dnia zapisane wielką literą	Sob
	J	Dzień w kalendarzu juliańskim — liczba dni, które minęły od 1 stycznia 4713 p.n.e.	2439892
Godzina	HH24	Dwie cyfry reprezentujące godzinę w formacie 24-godzinnym	23
	HH	Dwie cyfry reprezentujące godzinę w formacie 12-godzinnym	11
Minuta	MM	Dwie cyfry reprezentujące liczbę minut	57
Sekunda	SS	Dwie cyfry reprezentujące liczbę sekund	45

Tabela 5.2. Parametry formatujące daty i godziny — ciąg dalszy

Aspekt	Parametr	Opis	Przykład
	FF[1..9]	Ułamki sekund z opcjonalną liczbą cyfr po prawej stronie separatora dziesiętnego. Ten format ma zastosowanie wyłącznie w przypadku znaczników czasu, które zostały opisane w podrozdziale „Datowniki (znaczniki czasu)”	Jeżeli zastosujemy format FF3, liczba 0,123456789 sekund zostanie zaokrąglona do 0,123
	SSSS	Liczba sekund, które upłyнуły od godziny 00:00	46748
	MS	Milisekundy (milionowe części sekundy)	100
	CS	Centysekundy (setne części sekundy)	10
Separatory	- / , . ; : "tekst"	Znaki, które umożliwiają oddzielenie elementów dat i czasu. Jako separator można zastosować dowolny tekst, pod warunkiem że zostanie on umieszczony w cudzysłowie	Data 1 grudnia 1969 roku będzie reprezentowana w formacie DD-MM-YYYY jako 13-12-1969, a w formacie DD/MM/YYYY jako 13/12/1969
Sufiksy	AM lub PM	W zależności od pory dnia: PRZED. POŁ. lub PO. POŁ.	PRZED. POŁ.
	A.M. lub P.M.	W zależności od pory dnia: PRZED. POŁ. lub PO. POŁ.	PRZED. POŁ.
	AD lub BC	P.N.E. lub N.E.	P.N.E.
	A.D. lub B.C.	P.N.E. lub N.E.	P.N.E.
	TH	Przyrostek związany z odmianą liczby w języku angielskim (th, st, nd, rd, TH, ST, ND, RD). Jeżeli format liczby zostanie zapisany wielkimi literami, przyrostek również zostanie zapisany wielkimi literami. Jeżeli format liczby zostanie zapisany małymi literami, przyrostek również zostanie zapisany małymi literami	W przypadku dnia 28. ddTH wypisze 28 th , a DDTH wypisze 28TH. W przypadku dnia 21. ddTH wypisze 21 st , a DDTH wypisze 21ST. W przypadku dnia 22. ddTH wypisze 22 nd , a DDTH wypisze 22ND. W przypadku dnia 23. ddTH wypisze 23 rd , a DDTH wypisze 23RD.
	SP	Liczba jest wypisywana słownie w języku angielskim	W przypadku dnia 28. DDSP wypisze TWENTY-EIGHT, a ddSP — twenty-eight
	SPTH	Połączenie TH i SP	W przypadku dnia 28. DDSPTH wypisze TWENTY-EIGHTH, a ddSP — twenty-eighth
Era	EE	Pełna nazwa ery w przypadku kalendarza japońskiego i chińskiego oraz tajskiego kalendarza buddyjskiego	Brak przykładu
	E	Skrócona nazwa ery	Brak przykładu
Strefy czasowe	TZH	Godzina strefy czasowej.Więcej informacji o strefach czasowych znajduje się w podrozdziale „Strefy czasowe”	12
	TZM	Minuta strefy czasowej	30
	TZR	Nazwa strefy czasowej	PST
	TZD	Strefa czasowa z informacją o czasie letnim (zimowym)	Brak przykładu

W tabeli 5.3 przedstawiono przykłady napisów formatujących datę 5 lutego 1968 roku wraz z napisem zwracanym przez funkcję TO_CHAR().

Tabela 5.3. Przykładowe napisy formatujące datę i wyniki ich zastosowania

Napis formatujący	Zwracany napis
DD MONTH, YYYY	05 LUTY, 1968
DD/MM/YYYY	05/02/1968
DD-MM-YYYY	05-02-1968
MM/DD/YYYY	02/05/1968
DAY MON, YY AD	PONIEDZIAŁEK LUT, 68 N.E.
DDSPTH "of" MONTH, YEAR A.D.	FIFTH of LUTY, NINETEEN SIXTY-EIGHT N.E.
CC, SCC	20, 20
Q	1
YYYY, IYYY, RRRR, SYYYY, Y, YYY, YYY, IYY, YY, IY, RR, Y, I	1968, 1968, 1968, 1968, 1 968, 968, 968, 68, 68, 68, 8, 8
YEAR, Year	NINETEEN SIXTY-EIGHT, Nineteen Sixty-Eight
MM, MONTH, Month,	02, LUTY, Luty,
MON, Mon, RM	LUT, Lut, II
WW, IW, W	06, 06, 1
DD, DD, DAY	036, 05, PONIEDZIAŁEK
Day, DY, Dy, J	Poniedziałek, PON, Pon, 2439892
ddTH, DDTH, ddSP, DDSP, DDSPTH	05th, 05TH, five FIVE, FIFTH

Wyniki zamieszczone w tabeli można zobaczyć po wywołaniu funkcji TO_CHAR() w zapytaniu. Na przykład poniższe zapytanie konwertuje datę 5 lutego 1969 roku na napis o formacie MONTH DD, YYYY:

```
SELECT TO_CHAR(TO_DATE('68/02/05'), 'MONTH DD, YYYY')
FROM dual;
```

```
TO_CHAR(TO_DATE('68/02/05'), 'MONTH DD, YYYY')
```

```
-----
```

```
LUTY      05, 1968
```

 **Uwaga** Funkcja TO_DATE() konwertuje napis na wyrażenie *DataGodzina*. Zostanie ona opisana w kolejnym podrozdziale.

W tabeli 5.4 przedstawiono przykłady parametrów formatujących czas 19:32:36 (32 minuty i 36 sekund po 19) wraz z wynikiem zwróconym przez wywołanie funkcji TO_CHAR() przy takich danych wejściowych.

Tabela 5.4. Przykładowe napisy formatujące godzinę i wyniki ich zastosowania

Napis formatujący	Napis zwracany
HH24:MI:SS	19:32:36
HH.MI.SS AM	7.32.36 PO. POŁ.

Konwersja napisu na wyrażenie DataGodzina za pomocą funkcji TO_DATE()

Funkcja TO_DATE(*x* [, *format*]) konwertuje napis *x* na wyrażenie *DataGodzina*. Można przesyłać opcjonalny napis *format* definiujący format *x*. Jeżeli *format* zostanie pominięty, data musi zostać wprowadzona zgodnie z domyślnym formatem w bazie danych (w przypadku polskich ustawień narodowych YY/MM/DD).

W poniższym zapytaniu wykorzystano funkcję TO_DATE() do konwersji napisów 2007-LIP-04 i 07/07/04 na datę 4 lipca 2007 roku. Należy zwrócić uwagę, że w każdym przypadku zwrócona data jest wyświetlana w domyślnym formacie:

```
SELECT TO_DATE('2007-LIP-04'), TO_DATE('07/07/04')
FROM dual;
```

```
TO_DATE( TO_DATE(
-----
07/07/04 07/07/04
```

 Parametr NLS_DATE_FORMAT bazy danych określa domyślny format daty w bazie danych.

Jak zostanie to opisane w podrozdziale „Ustawianie domyślnego formatu daty”, ustawienia parametru NLS_DATE_FORMAT można zmieniać.

Określanie formatu daty i czasu

Jak wspomniano wcześniej, do funkcji TO_DATE() można przesłać opcjonalny parametr formatujący. Wykorzystywane są parametry formatujące opisane w tabeli 5.2. W poniższym zapytaniu użyto funkcji TO_DATE() do konwersji napisu 4 lipiec, 2012 na datę, przesyłając do funkcji TO_DATE() parametr formatujący DD_MONTH, YYYY:

```
SELECT TO_DATE('4 lipiec, 2007', 'DD MONTH, YYYY')
FROM dual;
```

```
TO_DATE(
-----
07/07/04
```

W kolejnym zapytaniu przesłano do funkcji TO_DATE() napis formatujący DD.MM.YY, aby przekonwertować na datę napis 4.7.07. Wynikowa data jest wyświetiana w domyślnym formacie YY/MM/DD:

```
SELECT TO_DATE('4.7.2007', 'DD.MM.YYYY')
FROM dual;
```

```
TO_DATE(
-----
07/07/04
```

Określanie czasu

W wyrażeniu *DataGodzina* możemy również określić godzinę. Jeżeli nie zostanie ona wpisana, baza przyjmuje domyślną wartość 0:00:00. Format czasu można określić za pomocą parametrów formatujących, przedstawionych wcześniej w tabeli 5.2. Jednym z przykładów takiego parametru jest HH24:MI:SS, gdzie:

Segment	Opis	Przykład
HH24	dwie cyfry godziny w formacie 24-godzinnym (od 00 do 23)	STYCZEN
MI	dwie cyfry minut (od 00 do 59)	02
SS	cztery cyfry sekund (od 00 do 59)	1965

Przykładem wyrażenia *DataGodzina* wykorzystującego format HH24:MI:SS jest 19:32:36. Pełnym przykładem wyrażenia *DataGodzina*, w którym określono czas, jest:

05-LUT-1968 19:32:36

gdzie format wyrażenia został zdefiniowany jako:

DD-MON-YYYY H24:MI:SS

Poniższe wywołanie funkcji TO_DATE() prezentuje wykorzystanie tego formatu i wartości:

```
TO_DATE('05-LUT-1968 19:32:36', 'DD-MON-YYYY HH24:MI:SS')
```

Wyrażenie *DataGodzina* zwracane przez funkcję TO_DATE() w poprzednim przykładzie zostało użyte w poniższej instrukcji INSERT do wstawienia wiersza do tabeli customers. Należy zwrócić uwagę, że w kolumnie dob nowy wiersz ma taką wartość jak wyrażenie *DataGodzina* zwracane przez funkcję TO_DATE():

```
INSERT INTO customers (
    customer_id, first_name, last_name,
    dob,
```

```

    phone
) VALUES (
  6, 'Fryderyk', 'Tytan',
  TO_DATE('05-LUT-1968 19:32:36', 'DD-MON-YYYY HH24:MI:SS'),
  '800-555-1215'
);

```

W celu wyświetlenia części godzinowej wyrażenia *DataGodzina* użyjemy funkcji `TO_CHAR()`. Na przykład poniższe zapytanie pobiera wiersze z tabeli `customers` i wykorzystuje funkcję `TO_CHAR()` do konwersji wartości z kolumny `dob`. Wartość dla klienta nr 6 jest taka, jak określiliśmy w instrukcji `INSERT`:

```

SELECT customer_id, TO_CHAR(dob, 'DD-MON-YYYY HH24:MI:SS')
FROM customers;

```

```

CUSTOMER_ID TO_CHAR(DOB, 'DD-MON-
-----
1 01-STY-1965 00:00:00
2 05-LUT-1968 00:00:00
3 16-MAR-1971 00:00:00
4
5 20-MAJ-1970 00:00:00
6 05-LUT-1968 19:32:36

```

W przypadku klientów nr 1, 2, 3 i 5 godzina w kolumnie `dob` ma wartość `00:00:00`. Jest to wartość domyślna, przyjmowana przez bazę danych, jeżeli godzina nie zostanie określona w wyrażeniu *DataGodzina*.

Kolejna instrukcja cofa wprowadzenie nowego wiersza:

```
ROLLBACK;
```

 **Uwaga** Jeżeli uruchomiono wcześniej przedstawioną instrukcję `INSERT`, należy wycofać zmiany za pomocą instrukcji `ROLLBACK`.

Łączenie wywołań funkcji `TO_CHAR()` i `TO_DATE()`

Wywołania funkcji `TO_CHAR()` i `TO_DATE()` można z sobą łączyć, co umożliwia wykorzystanie wyrażeń *DataGodzina* w różnych formatach. Na przykład w poniższym zapytaniu połączono wywołania funkcji `TO_CHAR()` i `TO_DATE()`, aby przejrzeć jedynie godzinową część wyrażenia *DataGodzina*. Wynik funkcji `TO_DATE()` jest przesyłany do funkcji `TO_CHAR()`:

```

SELECT TO_CHAR(TO_DATE('05-LUT-1968 19:32:36',
  'DD-MON-YYYY HH24:MI:SS'), 'HH24:MI:SS')
FROM dual;

```

```

TO_CHAR(
-----
19:32:36

```

Ustawianie domyślnego formatu daty

Domyślny format daty jest określany przez parametr `NLS_DATE_FORMAT` bazy danych. Administrator bazy danych może zmienić ustawienia tego parametru, wprowadzając odpowiedni wpis w pliku `init.ora` lub `spfile.ora`, które są odczytywane podczas uruchamiania bazy danych. Administrator może również ustawić wartość `NLS_DATE_FORMAT` za pomocą polecenia `ALTER SYSTEM`.

Użytkownik może również zmienić wartość `NLS_DATE_FORMAT` dla własnej sesji SQL*Plus za pomocą polecenia `ALTER SESSION`. Na przykład poniższa instrukcja `ALTER SESSION` przypisuje parametrowi `NLS_DATE_FORMAT` wartość `MONTH-DD-YYYY`:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'MONTH-DD-YYYY';
```

Session altered.



Uwaga Sesja rozpoczyna się podczas połączenia z bazą danych i jest zamkana w momencie rozłączenia.

Zastosowanie nowego formatu możemy zauważać w wynikach poniższego zapytania, pobierającego wartość kolumny dob dla klienta nr 1:

```
SELECT dob
FROM customers
WHERE customer_id = 1;
```

DOB

STYCZEŃ -01-1965

Nowy format może być również wykorzystany podczas wstawiania wiersza do bazy danych. Na przykład poniższa instrukcja INSERT wstawia nowy wiersz do tabeli customers. Należy zwrócić uwagę na zastosowanie formatu MONTH-DD-YYYY przy definiowaniu wartości dla kolumny dob:

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    6, 'Fryderyk', 'Tytan', 'MARZEC-15-1970', '800-555-1215'
);
```

Po rozłączeniu się z bazą danych i ponownym połączeniu zostanie przywrócony domyślny format daty. Dzieje się tak, ponieważ wszelkie zmiany wprowadzone za pomocą instrukcji ALTER SESSION mają zastosowanie tylko do bieżącej sesji — po zakończeniu połączenia zmiany nie są zachowywane.

Kolejna instrukcja cofa wprowadzenie nowego wiersza:

ROLLBACK;



Uwaga Jeżeli uruchomiono wcześniej przedstawioną instrukcję INSERT, należy wycofać zmiany za pomocą instrukcji ROLLBACK.

Aby przywrócić domyślne formatowanie w bieżącej sesji, należy wykonać następujące polecenie:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YY/MM/DD';
```

Session altered.

Jak Oracle interpretuje lata dwucyfrowe?

W bazie danych Oracle są składowane wszystkie cztery cyfry roku, ale jeżeli określmy tylko dwie z nich, baza inaczej zinterpretuje wiek w zależności od tego, czy użyjemy formatu YY, czy RR.



Wskazówka Należy zawsze określać wszystkie cztery cyfry roku. Pozwala to uniknąć pomyłek.

Najpierw zerkniemy na format YY, potem przejdziemy do formatu RR.

Użycie formatu YY

Jeżeli w stosowanym formacie dat lata są reprezentowane przez YY i wpiszemy tylko dwie cyfry roku, baza przyjmie, że wiek, w którym występuje ten rok, jest wiekiem aktualnie ustawionym w serwerze bazy danych. Dlatego też pierwsze dwie cyfry wprowadzonego roku są ustawiane na pierwsze dwie cyfry bieżącego roku.

Na przykład jeżeli wpiszemy 15, a mamy rok 2007, baza przyjmie, że chodzi nam o rok 2015. Jeżeli rok zdefiniujemy jako 75, zostanie on zapisany jako 2075.



Uwaga Jeżeli jest stosowany format YYYY, ale zostaną podane jedynie dwie cyfry roku, wprowadzone dane będą interpretowane zgodnie z formatem YY.

Przyjrzyjmy się zapytaniu, które wykorzystuje format YY do interpretacji lat 15 i 75. Wprowadzane daty — 15 i 75 — są przesyłane do funkcji TO_DATE(), której wynik jest z kolei przesyłany do funkcji TO_CHAR(), konwertującej daty na napisy zgodnie z formatem DD-MON-YYYY. Wykorzystywany jest tutaj format YYYY, aby widoczne były wszystkie cztery cyfry roku zwrócone przez funkcję TO_DATE():

```
SELECT
  TO_CHAR(TO_DATE('04-LIP-15', 'DD-MON-YY'), 'DD-MON-YYYY'),
  TO_CHAR(TO_DATE('04-LIP-75', 'DD-MON-YY'), 'DD-MON-YYYY')
FROM dual;
```

```
TO_CHAR(TO_DATE('04- TO_CHAR(TO_DATE('04-
-----
04-LIP-2015      04-LIP-2075
```

Zgodnie z oczekiwaniami lata 15 i 75 zostały zinterpretowane jako 2015 i 2075.

Użycie formatu RR

Jeżeli jest wykorzystywany format daty RR i wpiszemy dwie ostatnie cyfry roku, pierwsze dwie cyfry będą określane z użyciem wpisanych dwóch cyfr roku (*roku wprowadzonego*) i ostatnich dwóch cyfr roku skonfigurowanego na serwerze bazy danych (*roku bieżącego*). Reguły określania wieku przez bazę danych zostały opisane poniżej:

- **Reguła 1.** Jeżeli wprowadzony rok znajduje się w przedziale od 00 do 49 i bieżący rok również mieści się w tym przedziale, przyjmowany jest wiek bieżący, a więc *jako pierwsze dwie cyfry wprowadzonego roku są przyjmowane dwie pierwsze cyfry bieżącego roku*. Na przykład jeżeli w 2007 roku wprowadzimy 15, otrzymamy rok 2015.
- **Reguła 2.** Jeżeli wprowadzony rok znajduje się w przedziale od 50 do 99, a bieżący rok mieści się w przedziale od 00 do 49, przyjmowany jest wiek bieżący minus jeden, a więc *jako pierwsze dwie cyfry wprowadzonego roku są przyjmowane pierwsze dwie cyfry bieżącego roku minus jeden*. Na przykład jeżeli w 2007 roku wprowadzimy 75, otrzymamy rok 1975.
- **Reguła 3.** Jeżeli wprowadzony rok znajduje się w przedziale od 00 do 49, a bieżący rok znajduje się w przedziale od 50 do 99, przyjmowany jest wiek bieżący plus jeden, a więc *jako pierwsze dwie cyfry wprowadzonego roku są przyjmowane pierwsze dwie cyfry bieżącego roku plus jeden*. Na przykład jeżeli w 2075 roku wprowadzimy 15, otrzymamy rok 2115.
- **Reguła 4.** Jeżeli zarówno wprowadzony rok, jak i bieżący znajdują się w przedziale od 50 do 99, przyjmowany jest wiek bieżący, a więc *jako pierwsze dwie cyfry wprowadzonego roku są przyjmowane dwie pierwsze cyfry bieżącego roku*. Na przykład jeżeli w 2075 roku wprowadzimy 55, otrzymamy rok 2055.

Tabela 5.5 zawiera podsumowanie tych reguł.

Tabela 5.5. Sposób interpretacji lat dwucyfrowych

Wprowadzony rok dwucyfrowy			
	00 – 49	50 – 99	
Ostatnie dwie cyfry bieżącego roku	00 – 49	Reguła 1.: jako pierwsze dwie cyfry wprowadzonego roku przyjmowane są dwie pierwsze cyfry bieżącego roku	Reguła 2.: jako pierwsze dwie cyfry wprowadzonego roku przyjmowane są dwie pierwsze cyfry bieżącego roku minus jeden
	50 – 99	Reguła 3.: jako pierwsze dwie cyfry wprowadzonego roku przyjmowane są pierwsze dwie cyfry bieżącego roku plus jeden	Reguła 4.: jako pierwsze dwie cyfry wprowadzonego roku przyjmowane są dwie pierwsze cyfry bieżącego roku



Uwaga Jeżeli jest stosowany format RRRR, ale zostaną podane jedynie dwie cyfry roku, wprowadzone dane będą interpretowane zgodnie z formatem RR.

Przyjrzyjmy się zapytaniu, które wykorzystuje format YY do interpretacji lat 15 i 75. W poniższym zapytaniu należy założyć, że rokiem bieżącym jest 2012:

```
SELECT
  TO_CHAR(TO_DATE('04-LIP-15', 'DD-MON-RR'), 'DD-MON-YYYY') ,
  TO_CHAR(TO_DATE('04-LIP-75', 'DD-MON-RR'), 'DD-MON-YYYY')
FROM dual;
```

```
TO_CHAR(TO_DATE('04- TO_CHAR(TO_DATE('04-
```

```
-----  
04-LIP-2015      04-LIP-1975
```

Zgodnie z oczekiwaniami (reguła 1. i 2.) lata 15 i 75 zostały zinterpretowane jako 2015 i 1975.

W kolejnym przykładzie zakładamy, że bieżącym rokiem jest 2075:

```
SELECT
  TO_CHAR(TO_DATE('04-LIP-15', 'DD-MON-RR'), 'DD-MON-YYYY') ,
  TO_CHAR(TO_DATE('04-LIP-55', 'DD-MON-RR'), 'DD-MON-YYYY')
FROM dual;
```

```
TO_CHAR(TO_DATE('04- TO_CHAR(TO_DATE('04-
```

```
-----  
04-LIP-2115      04-LIP-2055
```

Zgodnie z oczekiwaniami (reguła 3. i 4.) lata 15 i 55 zostały zinterpretowane jako 2115 i 2055.

Funkcje operujące na datach i godzinach

Funkcje operujące na datach i godzinach są wykorzystywane do pobierania lub przetwarzania wyrażeń *DataGodzina* i znaczników czasu. W tabeli 5.6 opisano niektóre z tych funkcji. W tej tabeli x oznacza wyrażenie typu *DataGodzina* lub znacznik czasu.

Tabela 5.6. Funkcje operujące na datach i godzinach

Funkcja	Opis
ADD_MONTHS(x, y)	Zwraca wynik dodawania y miesięcy do x miesięcy. Jeżeli y jest liczbą ujemną, y miesiące jest odejmowane od x
LAST_DAY(x)	Zwraca ostatni dzień miesiąca będącego częścią wyrażenia x
MONTHS_BETWEEN(x, y)	Zwraca liczbę miesięcy między x i y. Jeżeli w kalendarzu x występuje przed y, jest zwracana liczba dodatnia. W przeciwnym razie jest zwracana liczba ujemna
NEXT_DAY(x, dzień)	Zwraca wyrażenie <i>DataGodzina</i> dla określonego dnia tygodnia występującego po x. Parametr <i>dzień</i> jest określany za pomocą literalu (na przykład SOBOTA)
ROUND(x [, jednostka])	Zaokrąglą x. Domyslnie x jest zaokrąglane do początku kolejnego dnia. Można przesyłać opcjonalny parametr <i>jednostka</i> , określający jednostkę zaokrąglenia. Na przykład YYYY zaokrąglą x do pierwszego dnia kolejnego roku
SYSDATE	Zwraca bieżącą datę i czas zdefiniowane w systemie operacyjnym serwera bazy danych
TRUNC(x [, jednostka])	Przycina x. Domyslnie x jest przycinane do początku dnia. Można przesyłać opcjonalny parametr <i>jednostka</i> definiujący jednostkę przycięcia. Na przykład MM przycina x do pierwszego dnia miesiąca

Funkcje przedstawione w tabeli 5.6 zostaną szczegółowo opisane w dalszych podrozdziałach.

ADD_MONTHS()

Funkcja ADD_MONTHS(*x,y*) zwraca wynik dodawania *y* miesięcy do *x*. Jeżeli *y* jest liczbą ujemną, *y* miesiące jest odejmowane od *x*.

W poniższym przykładzie dodano 13 miesięcy do dnia 1 stycznia 2012 roku:

```
SELECT ADD_MONTHS('01-STY-2012', 13)
FROM dual;
```

```
ADD_MONTHS(
-----
01-LUT-2013
```

W kolejnym przykładzie odjęto 13 miesięcy od dnia 13 stycznia 2008 roku. Należy zwrócić uwagę, że za pomocą funkcji ADD_MONTHS() do tej daty „dodano” –13 miesięcy:

```
SELECT ADD_MONTHS('01-STY-2008', -13)
FROM dual;
```

```
ADD_MONTHS(
-----
01-GRU-2010
```

Do funkcji ADD_MONTHS() można przesyłać datę i czas. Na przykład poniższe zapytanie dodaje dwa miesiące do wyrażenia *DataGodzina* 19:15:26 1 stycznia 2012 roku:

```
SELECT ADD_MONTHS(TO_DATE('01-STY-2012 19:15:26',
'DD-MON-YYYY HH24:MI:SS'), 2)
FROM dual;
```

```
ADD_MONTHS(
-----
01-MAR-2012
```

Kolejne zapytanie stanowi modyfikację powyższego w taki sposób, aby wyrażenie zwracane przez ADD_MONTHS() było konwertowane za pomocą funkcji TO_CHAR() na napis w formacie DD-MON-YYYY HH24:MI:SS:

```
SELECT TO_CHAR(ADD_MONTHS(TO_DATE('01-STY-2012 19:15:26',
'DD-MON-YYYY HH24:MI:SS'), 2), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;
```

```
TO_CHAR(ADD_MONTHS(T
-----
01-MAR-2012 19:15:26
```



Wyrażenie *DataGodzina* można przesyłać do każdej funkcji przedstawionej w tabeli 5.6.

LAST_DAY()

Funkcja LAST_DAY(*x*) zwraca ostatni dzień miesiąca będącego częścią wyrażenia *x*. Poniższe zapytanie pobiera datę ostatniego dnia stycznia 2008 roku:

```
SELECT LAST_DAY('01-STY-2008')
FROM dual;
```

```
LAST_DAY('0
-----
31-STY-2008
```

MONTHS_BETWEEN()

Funkcja MONTHS_BETWEEN(*x, y*) zwraca liczbę miesięcy między *x* i *y*. Jeżeli w kalendarzu *x* występuje przed *y*, jest zwracana liczba dodatnia.



Kolejność dat w wywołaniu funkcji `MONTHS_BETWEEN()` jest ważna. Jeżeli zwrócona liczba ma być dodatnia, późniejsza data musi być przesłana jako pierwsza.

Poniższy przykład oblicza liczbę miesięcy między 25 maja i 15 stycznia 2012 roku. Należy zauważyć, że ponieważ późniejsza data została przesłana jako pierwsza, została zwrócona liczba dodatnia:

```
SELECT MONTHS_BETWEEN('25-MAJ-2012', '15-STY-2012')
FROM dual;
```

```
MONTHS_BETWEEN('25-MAJ-2012','15-STY-2012')
-----
4,32258065
```

W poniższym przykładzie odwrócono kolejność dat w wywołaniu funkcji `MONTHS_BETWEEN()`, dlatego została zwrócona ujemna liczba miesięcy:

```
SELECT MONTHS_BETWEEN('15-STY-2012', '25-MAJ-2012')
FROM dual;
```

```
MONTHS_BETWEEN('15-STY-2012','25-MAJ-2012')
-----
-4,3225806
```

NEXT_DAY()

Funkcja `NEXT_DAY(x, dzień)` zwraca datę dla określonego dnia tygodnia występującego po `x`. Parametr `dzień` jest określany za pomocą literału (na przykład `SOBOTA`).

Poniższy przykład wyświetla datę pierwszej soboty po 1 stycznia 2012 roku:

```
SELECT NEXT_DAY('01-STY-2012', 'SOBOTA')
FROM DUAL;
```

```
NEXT_DAY('0
-----
05-STY-2012
```

ROUND()

Funkcja `ROUND(x [, jednostka])` zaokrąglą domyślnie `x` do początku kolejnego dnia. Można przesłać opcjonalny parametr `jednostka`, określający jednostkę zaokrąglenia. Na przykład `YYYY` zaokrąglą `x` do pierwszego dnia kolejnego roku. Do określenia jednostki zaokrąglania można użyć wielu parametrów spośród przedstawionych w tabeli 5.2.

W poniższym przykładzie użyto funkcji `ROUND()` do zaokrąglenia 25 października 2012 do pierwszego dnia kolejnego roku, czyli 1 stycznia 2013. Należy zauważyć, że data jest określona jako `25-PAŹ-2012` i umieszczona w wywołaniu funkcji `TO_DATE()`:

```
SELECT ROUND(TO_DATE('25-PAŹ-2012'), 'YYYY')
FROM dual;
```

```
ROUND(TO_DA
-----
01-STY-2013
```

Kolejny przykład zaokrąglą datę 25 maja 2012 roku do pierwszego dnia najbliższego miesiąca, czyli do 1 czerwca 2012, ponieważ 25 maja jest bliżej początku czerwca niż początku maja.

```
SELECT ROUND(TO_DATE('25-MAJ-2012'), 'MM')
FROM dual;
```

```
ROUND(TO_DA
-----
01-CZE-2012
```

Kolejny przykład zaokrąglą czas 19:45:26 w dniu 25 maja 2012 roku do najbliższej pełnej godziny, czyli 20:00:

```
SELECT TO_CHAR(ROUND(TO_DATE('25-MAJ-2012 19:45:26',
  'DD-MON-YYYY HH24:MI:SS'), 'HH24'), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;
```

```
TO_CHAR(ROUND(TO_DAT
-----
25-MAJ-2012 20:00:00
```

SYSDATE

SYSDATE zwraca bieżącą datę i czas ustawione w systemie operacyjnym serwera bazy danych. Poniższe pytanie pobiera bieżącą datę:

```
SELECT SYSDATE
FROM dual;
```

```
SYSDATE
-----
05-LIS-2007
```

TRUNC()

Funkcja `TRUNC(x [, jednostka])` przycina `x`. Domyślnie `x` jest przycinane do początku dnia. Można przesłać opcjonalny parametr `jednostka`, definiujący jednostkę przycięcia. Na przykład `MM` przycina `x` do pierwszego dnia miesiąca. Do przycinania `x` można użyć wielu parametrów spośród przedstawionych w tabeli 5.2.

W poniższym przykładzie użyto funkcji `TRUNC()` do przycięcia daty 25 maja 2012 do pierwszego dnia roku, czyli do 1 stycznia 2012:

```
SELECT TRUNC(TO_DATE('25-MAJ-2012'), 'YYYY')
FROM dual;
```

```
TRUNC(TO_DA
-----
01-STY-2012
```

Kolejny przykład przycina datę 25 maja 2012 roku do pierwszego dnia miesiąca, czyli 1 maja 2012:

```
SELECT TRUNC(TO_DATE('25-MAJ-2012'), 'MM')
FROM dual;
```

```
TRUNC(TO_DA
-----
01-MAJ-2012
```

Kolejny przykład przycina godzinę 19:45:26 w dniu 25 maja 2012 roku do pełnej godziny, czyli 19:00:

```
SELECT TO_CHAR(TRUNC(TO_DATE('25-MAJ-2012 19:45:26',
  'DD-MON-YYYY HH24:MI:SS'), 'HH24'), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;
```

```
TO_CHAR(TRUNC(TO_DAT
-----
25-MAJ-2012 19:00:00
```

Strefy czasowe

W Oracle Database 9i wprowadzono możliwość ustawiania różnych stref czasowych. Strefy czasowe określają się względem czasu w Greenwich. Był on niegdyś nazywany czasem uniwersalnym (*Greenwich Mean Time*, GMT), ale teraz określa się go mianem uniwersalnego czasu skoordynowanego (UTC).

Strefę czasową określa się, podając albo przesunięcie od UTC, albo region geograficzny (na przykład PST). Podając przesunięcie, stosuje się format HH:MI poprzedzony znakiem plus lub minus:

+/-HH:MI

gdzie:

- + lub – określa dodatnie albo ujemne przesunięcie względem UTC,
- HH:MI określa przesunięcie w godzinach i minutach.



Uwaga Godziny i minuty strefy czasowej wykorzystują parametry formatujące TZH i TZR, prezentowane w tabeli 5.2.

Poniższy przykład przedstawia przesunięcia o 8 godzin przed UTC oraz 2 godziny i 15 minut po UTC:

-08:00
+02:15

Strefę czasową można również określić za pomocą regionu geograficznego. Na przykład PST określa standardowy czas Pacyfiku (*Pacific Standard Time*), różniący się o -8 godzin od UTC. EST określa standardowy czas wschodni (*Eastern Standard Time*), różniący się o -5 od UTC.



Uwaga Region strefy czasowej wykorzystuje parametr formatujący TZR, prezentowany w tabeli 5.2.

Funkcje operujące na strefach czasowych

Istnieje wiele funkcji związanych ze strefami czasowymi. Są one przedstawione w tabeli 5.7. Funkcje te zostaną szczegółowo opisane w kolejnych podrozdziałach.

Tabela 5.7. Funkcje operujące na strefach czasowych

Funkcja	Opis
CURRENT_DATE	Zwraca bieżącą datę w lokalnej strefie czasowej ustawionej dla sesji bazy danych
DBTIMEZONE	Zwraca strefę czasową bazy danych
NEW_TIME(x, strefa_czasowa1, strefa_czasowa2)	Konwertuje x ze strefy czasowej <i>strefa_czasowa1</i> na <i>strefa_czasowa2</i> i zwraca nową datę i czas
SESSIONTIMEZONE	Zwraca strefę czasową dla sesji bazy danych
TZ_OFFSET(<i>strefa_czasowa</i>)	Zwraca przesunięcie dla strefy <i>strefa_czasowa</i> w godzinach i minutach

Strefa czasowa bazy danych i strefa czasowa sesji

Jeżeli pracujemy w dużej organizacji, baza danych, do której uzyskujemy dostęp, może znajdować się w innej strefie czasowej niż ta, w której aktualnie przebywamy. Strefę czasową dla bazy danych nazywamy **strefą czasową bazy danych**, a strefę ustawioną dla sesji — **strefą czasową sesji**. Zostaną one opisane w kolejnych podrozdziałach.

Strefa czasowa bazy danych

Strefa czasowa bazy danych jest ustawiana za pomocą parametru TIME_ZONE bazy danych. Administrator bazy danych może zmienić wartość tego parametru w plikach konfiguracyjnych *init.ora* lub *spfile.ora* albo za pomocą polecenia:

`ALTER DATABASE SET TIME_ZONE = przesunięcie | region`

Na przykład:

```
ALTER DATABASE SET TIME_ZONE = '8:00'  
ALTER DATABASE SET TIME_ZONE = 'PST'
```

Strefę czasową ustawioną dla bazy danych możemy wyświetlić za pomocą funkcji DBTIMEZONE. Na przykład poniższe polecenie pobiera strefę czasową ustawioną w bazie danych autora:

```
SELECT DBTIMEZONE
FROM dual;
```

```
DBTIME
-----
+00:00
```

Jak widzimy, zapytanie zwróciło +00:00, co oznacza, że w bazie danych jest wykorzystywana strefa czasowa ustawiona w systemie operacyjnym.



System operacyjny Windows zazwyczaj dostosowuje ustawienia zegara przy zmianie czasu z zimowego na letni. W Polsce czas letni jest przesunięty względem UTC o dwie godziny do przodu, a czas zimowy — o godzinę. W przykładach z tego rozdziału strefa czasowa jest przesunięta względem UTC o dwie godziny do przodu.

Strefa czasowa sesji

Strefa czasowa sesji jest ustawiana dla konkretnej sesji. Domyślnie jest taka sama jak strefa czasowa ustawiona w systemie operacyjnym. Można ją zmienić, wprowadzając inną wartość parametru TIME_ZONE bieżącej sesji za pomocą instrukcji ALTER SESSION. Na przykład kolejne wyrażenie ustawia lokalną strefę czasową na czas środkowoeuropejski:

```
ALTER SESSION SET TIME_ZONE = 'CET'
```

Można również ustawić parametr TIME_ZONE na LOCAL, co zmieni strefę czasową sesji na ustawioną w systemie operacyjnym komputera, z którego została uruchomiona instrukcja ALTER SESSION. Można również ustawić strefę sesji (parametr TIME_ZONE) na DBTIMEZONE, czyli na strefę czasową bazy danych.

Strefę czasową ustawioną dla bazy danych możemy wyświetlić za pomocą funkcji SESSIONTIMEZONE. Poniższe zapytanie pobiera strefę czasową sesji:

```
SELECT SESSIONTIMEZONE
FROM dual;
```

```
SESSIONTIMEZONE
-----
+02:00
```

Lokalna strefa czasowa jest przesunięta o dwie godziny do przodu względem UTC.

Pobieranie bieżącej daty w strefie czasowej sesji

Funkcja SYSDATE zwraca datę z bazy danych. Uzyskujemy dzięki temu datę w strefie czasowej bazy danych. Datę ze strefy czasowej sesji można uzyskać za pomocą funkcji CURRENT_DATE. Na przykład:

```
SELECT CURRENT_DATE
FROM dual;
```

```
CURRENT_
-----
08/09/29
```

Uzyskiwanie przesunięć strefy czasowej

Przesunięcie strefy czasowej wyrażone w godzinach możemy uzyskać, przesyłając do funkcji TZ_OFFSET() nazwę regionu. Na przykład poniższe zapytanie wykorzystuje funkcję TZ_OFFSET() do pobrania przesunięcia strefy czasowej PST, czyli UTC -8:00 (w lecie UTC -7:00):

```
SELECT TZ_OFFSET('PST')
FROM dual;
```

TZ_OFFSET

-07:00



W systemie Windows w zimie będzie to -8:00, ponieważ uwzględnia on automatycznie czas zimowy.

Uzyskiwanie nazw stref czasowych

Możemy uzyskać wszystkie nazwy stref czasowych, pobierając wszystkie wiersze z tabeli v\$timezone_names. Aby przesłać zapytanie do v\$timezone_name, należy połączyć się z bazą danych jako użytkownik system. Poniżej zapytanie pobiera pięć pierwszych wierszy z tabeli v\$timezone_names:

```
SELECT * FROM v$timezone_names
WHERE ROWNUM <=5 ORDER BY tzabbrev;
```

TZNAME	TZABBREV
Africa/Algiers	CET
Africa/Algiers	LMT
Africa/Algiers	PMT
Africa/Algiers	WEST
Africa/Algiers	WET

Do ustawiania stref czasowych można użyć dowolnej wartości tzname lub tzabbrev.



Pseudokolumna ROWNUM zawiera numer wiersza. Na przykład pierwszy wiersz zwrocony przez zapytanie był wierszem nr 1, drugi miał nr 2 itd., dlatego też klauzula WHERE w powyższym zapytaniu powoduje, iż zwraca ono pierwsze pięć wierszy tabeli.

Konwertowanie wyrażenia DataGodzina z jednej strefy czasowej na inną

Do konwersji wyrażenia *DataGodzina* z jednej strefy czasowej na inną służy funkcja NEW_TIME(). Na przykład poniżejsze zapytanie wykorzystuje ją do konwersji godziny 19:45 13 maja 2012 roku ze strefy PST na EST:

```
SELECT TO_CHAR(NEW_TIME(TO_DATE('13-MAJ-2012 19:45',
  'DD-MON-YYYY HH24:MI'), 'PST', 'EST'), 'DD-MON-YYYY HH24:MI')
FROM dual;
```

```
TO_CHAR(NEW_TIME(
-----
13-MAJ-2012 22:45
```

Strefa EST jest przesunięta o trzy godziny do przodu względem PST, więc po dodaniu tych trzech godzin do 19:45 otrzymujemy 22:45.

Datowniki (znaczniki czasu)

W Oracle Database 9i wprowadzono możliwość składowania datowników (TIMESTAMP). Datownik przechowuje wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minuty, sekundy, ułamkowe części sekund i strefę czasową. Znaczniki czasu mają następujące zalety w porównaniu do typu DATA:

- mogą przechowywać ułamki sekund,
- mogą przechowywać strefę czasową.

Datowniki zostaną opisane w kolejnych podrozdziałach.

Typy datowników

Występują trzy typy datowników. Zostały one przedstawione w tabeli 5.8. Sposoby użycia wymienionych datowników zostaną opisane w kolejnych podrozdziałach.

Tabela 5.8. Typy datowników

Typ	Opis
<code>TIMESTAMP[(precyzja_sekund)]</code>	Przechowuje wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym) minuty oraz sekundy. Można określić precyzję dla sekund, przesyłając parametr <i>precyzja_sekund</i> , który może być liczbą całkowitą od 0 do 9, co oznacza, że możemy przechować do 9 cyfr po prawej stronie separatora. Jeżeli spróbujemy dodać wiersz zawierający w ułamkowej części sekund więcej cyfr, niż może pomieścić typ <code>TIMESTAMP</code> , ułamek zostanie zaokrąglony
<code>TIMESTAMP[(precyzja_sekund)] WITH TIME ZONE</code>	Rozszerza typ <code>TIMESTAMP</code> o przechowywanie strefy czasowej
<code>TIMESTAMP[(precyzja_sekund)] WITH LOCAL TIME ZONE</code>	Rozszerza typ <code>TIMESTAMP</code> o konwersję przesłanej daty i czasu do lokalnej strefy czasowej bazy danych. Tę konwersję nazywamy normalizacją daty i czasu

Użycie typu `TIMESTAMP`

Podobnie jak w przypadku innych typów danych możemy użyć typu `TIMESTAMP` do zdefiniowania kolumny w tabeli. Poniższa instrukcja tworzy tabelę o nazwie `purchases_with_timestamp`, w której będą przechowywane informacje o zakupach dokonanych przez klientów. Ta tabela zawiera kolumnę `made_on` z typem `TIMESTAMP`, w której będzie zapisywany moment dokonania zakupu. Należy zauważyć, że dla typu `TIMESTAMP` ustalono precyzję 4 (to oznacza, że sekundy będą przechowywane z dokładnością do czwartego miejsca po przecinku):

```
CREATE TABLE purchases_with_timestamp (
    product_id INTEGER REFERENCES products(product_id),
    customer_id INTEGER REFERENCES customers(customer_id),
    made_on TIMESTAMP(4)
);
```

 Skrypt `store_schema.sql` tworzy tabelę `purchases_with_timestamp` i umieszcza w niej dane. Pozostałe tabele prezentowane w tym rozdziale są również tworzone przez ten skrypt, więc nie jest konieczne wpisywanie instrukcji `CREATE TABLE` i `INSERT`.

Aby przesyłać literal o typie `TIMESTAMP`, należy użyć słowa kluczowego `TIMESTAMP` oraz wyrażenia *Data-Godzina* w formacie:

`TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.SSSSSSS'`

Należy zauważyć, że po przecinku znajduje się dziewięć znaków `S`, co oznacza, że w literale może znajdować się do dziewięciu cyfr po przecinku dla ułamków sekundy. Liczba cyfr, które zostaną faktycznie przechowane w kolumnie `TIMESTAMP`, zależy od tego, jak została zdefiniowana kolumna. Na przykład w kolumnie `made_on` tabeli `purchases_with_timestamp` może być zapisanych do czterech cyfr po przecinku. Jeżeli spróbujemy dodać wiersz zawierający więcej cyfr po przecinku, część ułamkowa zostanie zaokrąglona. Na przykład:

2005-05-13 07:15:31.123456789

zostanie zaokrąglone do:

2005-05-13 07:15:31:123

Poniższa instrukcja `INSERT` wstawia wiersz do tabeli `purchases_with_timestamp`. Należy zauważyć, że użyto słowa kluczowego `TIMESTAMP` do przesłania literala:

```
INSERT INTO purchases_with_timestamp (
    product_id, customer_id, made_on
) VALUES (
    1, 1, TIMESTAMP '2005-05-13 07:15:31.1234'
);
```

Poniższe zapytanie pobiera wiersz:

```
SELECT *
FROM purchases_with_timestamp;

PRODUCT_ID CUSTOMER_ID MADE_ON
-----
1           1 05/05/13 07:15:31,1234
```

Użycie typu TIMESTAMP WITH TIME ZONE

Typ **TIMESTAMP WITH TIME ZONE** rozszerza typ **TIMESTAMP** o przechowywanie strefy czasowej. Poniższa instrukcja tworzy tabelę o nazwie **purchases_timestamp_with_tz**, w której są zapisywane dane o zakupach dokonanych przez klientów. Tabela zawiera kolumnę **made_on** typu **TIMESTAMP WITH TIME ZONE**, w której jest rejestrowany moment zakupu:

```
CREATE TABLE purchases_timestamp_with_tz (
    product_id INTEGER REFERENCES products(product_id),
    customer_id INTEGER REFERENCES customers(customer_id),
    made_on TIMESTAMP(4) WITH TIME ZONE
);
```

Aby przesyłać do bazy danych literal datownika ze strefą czasową, wystarczy dodać strefę czasową do typu **TIMESTAMP**. Na przykład poniższy typ **TIMESTAMP** zawiera przesunięcie strefy czasowej wynoszące -07:00: **TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'**

Można również przesyłać region strefy czasowej, co przedstawia poniższy przykład, w którym jako strefę czasową określono PST:

```
TIMESTAMP '2005-05-13 07:15:31.1234 PST'
```

Poniższy przykład wstawia dwa wiersze do tabeli **purchases_timestamp_with_tz**:

```
INSERT INTO purchases_timestamp_with_tz (
    product_id, customer_id, made_on
) VALUES (
    1, 1, TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'
);

INSERT INTO purchases_timestamp_with_tz (
    product_id, customer_id, made_on
) VALUES (
    1, 2, TIMESTAMP '2005-05-13 07:15:31.1234 PST'
);
```

Poniższe zapytanie pobiera te wiersze:

```
SELECT *
FROM purchases_timestamp_with_tz;

PRODUCT_ID CUSTOMER_ID MADE_ON
-----
1           1 05/05/13 07:15:31,1234 -07:00
1           2 05/05/13 07:15:31,1234 PST
```

Użycie typu TIMESTAMP WITH LOCAL TIME ZONE

Typ **TIMESTAMP WITH LOCAL TIME ZONE** rozszerza typ **TIMESTAMP** i umożliwia automatyczne konwertowanie czasu do lokalnej strefy czasowej ustawionej dla bazy danych. Po przesłaniu datownika do zapisania w kolumnie typu **TIMESTAMP WITH LOCAL TIME ZONE** jest on konwertowany — czy też *normalizowany* — do

strefy czasowej ustawionej dla bazy danych. Przy pobieraniu znacznika czasu jest on normalizowany do strefy czasowej ustawionej dla sesji.



Wskazówka Typ `TIMESTAMP WITH LOCAL TIME ZONE` powinien być używany, jeżeli organizacja posiada system wykorzystywany w różnych częściach świata. Przechowuje on datownik, korzystając z lokalnej strefy czasowej obowiązującej w miejscu, w którym znajduje się baza danych, ale do użytkowników są przesyłane datowniki przystosowane do ich strefy czasowej.

Załóżmy, że w bazie danych jest ustawiona strefa czasowa CET (czyli UTC + 2:00) i chcemy w niej zapisać następujący datownik:

2005-05-13 07:15:30 EST

Strefa EST jest określana jako UTC – 5:00, więc różnica między CET i EST wynosi 7 godzin ($5 + 2 = 7$). Dlatego też w celu normalizacji powyższego znacznika czasu do CET należy dodać do niego 7 godzin, dzięki czemu uzyskamy odpowiedni datownik:

2005-05-13 14:15:30

Właśnie taki znacznik czasu zostanie zapisany w kolumnie typu `TIMESTAMP WITH LOCAL TIME ZONE`.

Poniższa instrukcja tworzy tabelę `purchases_with_local_tz`, w której są zapisywane informacje o zakupach dokonanych przez klientów. Tabela zawiera kolumnę `made_on` typu `TIMESTAMP WITH LOCAL TIME ZONE`, w której jest rejestrowany moment zakupu:

```
CREATE TABLE purchases_with_local_tz (
    product_id INTEGER REFERENCES products(product_id),
    customer_id INTEGER REFERENCES customers(customer_id),
    made_on TIMESTAMP(4) WITH LOCAL TIME ZONE
);
```

Poniższa instrukcja `INSERT` wstawia do tabeli `purchases_with_local_tz` wiersz, w którym pole `made_on` ma wartość 2005-05-13 07:15:30 EST:

```
INSERT INTO purchases_with_local_tz (
    product_id, customer_id, made_on
) VALUES (
    1, 1, TIMESTAMP '2005-05-13 07:15:30 EST'
);
```

Choć datownik przesłany do kolumny `made_on` ma wartość 2005-05-13 07:15:30 EST, w opisywanej bazie danych zostanie zapisana wartość 2005-05-13 14:15:30 (znacznik czasu znormalizowany dla strefy CET).

Poniższe zapytanie pobiera ten wiersz:

```
SELECT *
FROM purchases_with_local_tz;
-----  
PRODUCT_ID CUSTOMER_ID MADE_ON  
-----  
1           1 05/05/13 14:15:30,0000
```

Ponieważ w opisywanej bazie danych strefa czasowa zarówno bazy danych, jak i sesji są ustawione na CET, datownik zwrócony przez zapytanie ma wartość dla tej strefy.



Uwaga Znacznik czasu zwrócony przez powyższe zapytanie jest znormalizowany dla strefy CET. Jeżeli strefa czasowa bazy danych lub sesji jest inna niż CET, datownik zwrócony przez zapytanie będzie miał inną wartość (będzie znormalizowany dla obowiązującej strefy czasowej).

Gdybyśmy ustawiли strefę czasową sesji na EST i wykonali ponownie powyższe zapytanie, zostałyby zwrócony znacznik znormalizowany dla tej strefy:

```
ALTER SESSION SET TIME_ZONE = 'EST';
```

Session altered.

```
SELECT *
FROM purchases_with_local_tz;

PRODUCT_ID CUSTOMER_ID MADE_ON
----- -----
1          1 05/05/13 07:15:30,0000
```

Znacznik zwrócony przez zapytanie ma wartość 05/05/13 07:15:30,0000, czyli jest znormalizowany dla strefy czasowej sesji (EST). Ponieważ w strefie EST czas jest przesunięty o siedem godzin do tyłu względem CET, od wartości 05/05/13 14:15:30,0000 (znacznika składowanego w bazie danych) musi zostać odjętych siedem godzin.

Poniższa instrukcja przywraca dla sesji strefę czasową CET:

```
ALTER SESSION SET TIME_ZONE = 'CET';
```

Session altered.

Funkcje operujące na znacznikach czasu

Istnieje kilka funkcji umożliwiających pobieranie i przetwarzanie datowników. Zostały one przedstawione w tabeli 5.9. Funkcje te zostaną szczegółowo opisane w kolejnych podrozdziałach.

Tabela 5.9. Funkcje operujące na datownikach

Funkcja	Opis
CURRENT_TIMESTAMP	Zwraca wartość typu TIMESTAMP WITH TIME ZONE, zawierającą bieżącą datę i czas w sesji oraz strefę czasową sesji
EXTRACT({ YEAR MONTH DAY HOUR MINUTE SECOND } { TIMEZONE_HOUR TIMEZONE_MINUTE } { TIMEZONE_REGION } TIMEZONE_ABBR) FROM x)	Wydobywa i zwraca rok, miesiąc, dzień, godzinę, minuty, sekundy lub strefę czasową z x. Wartość x może być datownikiem lub typem DATE
FROM_TZ(x, strefa_czasowa)	Konwertuje x typu TIMESTAMP do strefy czasowej określonej przez parametr strefa_czasowa i zwraca typ TIMESTAMP WITH TIMEZONE. Parametr strefa_czasowa musi być określony jako napis w postaci + - HH:MI.
LOCALTIMESTAMP	Funkcja po prostu scala x i strefa_czasowa w jedną wartość
SYSTIMESTAMP	Zwraca typ TIMESTAMP, zawierający bieżącą datę i czas sesji
SYS_EXTRACT_UTC(x)	Zwraca typ TIMESTAMP WITH TIMEZONE, zawierający bieżącą datę i czas UTC
TO_TIMESTAMP(x, [format])	Konwertuje x typu TIMESTAMP WITH TIMEZONE na typ TIMESTAMP, zawierający datę i czas UTC
TO_TIMESTAMP_TZ(x, [format])	Konwertuje napis x na typ TIMESTAMP. Można również przesyłać opcjonalny parametr format dla x
	Konwertuje napis x na typ TIMESTAMP WITH TIMEZONE. Można również przesyłać opcjonalny parametr format dla x

CURRENT_TIMESTAMP, LOCALTIMESTAMP i SYSTIMESTAMP

Poniższe zapytanie wywołuje funkcje CURRENT_TIMESTAMP, LOCALTIMESTAMP i SYSTIMESTAMP (strefa czasowa bazy danych i sesji to CET, czyli dwie godziny do przodu względem UTC):

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP
FROM dual;
```

```
CURRENT_TIMESTAMP
-----
LOCALTIMESTAMP
```

```
-----  
SYSTIMESTAMP
```

```
11/09/30 15:46:26,765000 +02:00  
11/09/30 15:46:26,765000 +02:00  
11/09/30 15:46:26,765000 +02:00
```

Jeżeli zmienimy wartość parametru TIME_ZONE sesji na EST i powtórzmy poprzednie zapytanie, uzyskamy następujące wyniki:

```
ALTER SESSION SET TIME_ZONE = 'EST';
```

Session altered.

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP  
FROM dual;
```

```
CURRENT_TIMESTAMP
```

```
LOCALTIMESTAMP
```

```
-----  
SYSTIMESTAMP
```

```
11/09/30 08:47:59,968000 EST  
11/09/30 08:47:59,968000  
11/09/30 15:47:59,968000 +02:00
```

Poniższa instrukcja przywraca strefę czasową CET dla sesji:

```
ALTER SESSION SET TIME_ZONE = 'CET';
```

Sesja została zmieniona.

EXTRACT()

Funkcja EXTRACT() wydobywa i zwraca rok, miesiąc, dzień, godzinę, minuty, sekundy lub strefę czasową z x. Wartość x może być datownikiem lub typem DATE. Poniższe zapytanie wykorzystuje funkcję EXTRACT() do pobrania roku, miesiąca i dnia z typu DATE zwróconego przez funkcję TO_DATE():

```
SELECT  
    EXTRACT(YEAR FROM TO_DATE('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS ROK,  
    EXTRACT(MONTH FROM TO_DATE('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS MIESIĄC,  
    EXTRACT(DAY FROM TO_DATE('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS DZIEŃ  
FROM dual;
```

ROK	MIESIĄC	DZIEŃ
2012	1	1

Kolejne zapytanie wykorzystuje funkcję EXTRACT() do pobrania godziny, minut i sekund z typu TIMESTAMP zwróconego przez funkcję TO_TIMESTAMP:

```
SELECT  
    EXTRACT(HOUR FROM TO_TIMESTAMP('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS GODZINA,  
    EXTRACT(MINUTE FROM TO_TIMESTAMP('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS MINUTY,  
    EXTRACT(SECOND FROM TO_TIMESTAMP('01-STY-2012 19:15:26',  
        'DD-MON-YYYY HH24:MI:SS')) AS SEKUNDY  
FROM dual;
```

GODZINA	MINUTY	SEKUNDY
19	15	26

Poniższe zapytanie wykorzystuje funkcję EXTRACT() do pobrania godziny, minut, sekund, regionu oraz skrótu nazwy regionu strefy czasowej z typu TIMESTAMP WITH TIMEZONE zwróconego przez funkcję TO_TIMESTAMP_TZ():

```
SELECT
  EXTRACT(TIMEZONE_HOUR FROM TO_TIMESTAMP_TZ(
    '01-STY-2012 19:15:26 -7:15', 'DD-MON-YYYY HH24:MI:SS TZH:TZM'))
  AS TZH,
  EXTRACT(TIMEZONE_MINUTE FROM TO_TIMESTAMP_TZ(
    '01-STY-2012 19:15:26 -7:15', 'DD-MON-YYYY HH24:MI:SS TZH:TZM'))
  AS TZM,
  EXTRACT(TIMEZONE_REGION FROM TO_TIMESTAMP_TZ(
    '01-STY-2012 19:15:26 CET', 'DD-MON-YYYY HH24:MI:SS TZR'))
  AS TZR,
  EXTRACT(TIMEZONE_ABBR FROM TO_TIMESTAMP_TZ(
    '01-STY-2012 19:15:26 CET', 'DD-MON-YYYY HH24:MI:SS TZR'))
  AS TZA
FROM dual;
```

TZH	TZM	TZR	TZA
-7	-15	CET	CET

FROM_TZ()

Funkcja FROM_TZ(*x, strefa_czasowa*) konwertuje wartość *x* typu TIMESTAMP na strefę czasową określoną przez parametr *strefa_czasowa* i zwraca typ TIMESTAMP WITH TIMEZONE. Parametr *strefa_czasowa* musi być określony jako napis w postaci +|- HH:MI. Funkcja po prostu scala *x* i *strefa_czasowa* w jedną wartość.

Na przykład poniższe zapytanie scala datownik 2012-05-13 07:15:31.1234 i przesunięcie strefy czasowej o -7:00 względem UTC:

```
SELECT FROM_TZ(TIMESTAMP '2012-05-13 07:15:31.1234', '-7:00')
FROM dual;
```

```
FROM_TZ(TIMESTAMP'2012-05-1307:15:31.1234','-7:00')
```

```
12/05/13 07:15:31,123400000 -07:00
```

SYS_EXTRACT_UTC()

Funkcja SYS_EXTRACT_UTC() konwertuje wartość *x* typu TIMESTAMP WITH TIMEZONE na typ TIMESTAMP zawierający datę i czas w UTC.

Poniższe zapytanie konwertuje datownik 2012-11-17 19:15:26 CET na UTC:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2012-11-17 19:15:26 CET')
FROM dual;
```

```
SYS_EXTRACT_UTC(TIMESTAMP'2012-11-1719:15:26CET')
```

```
12/11/17 18:15:26,000000000
```

Ponieważ w zimie czas w strefie CET jest przesunięty o godzinę do przodu względem UTC, zapytanie zwróciło znacznik przesunięty o godzinę do tyłu względem 2012-11-17 19:15:26 CET, czyli 2012-11-17 18:15:26.

W przypadku daty letniej zwrócony datownik będzie przesunięty o dwie godziny do tyłu względem UTC:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2012-05-17 19:15:26 CET')
FROM dual;
```

```
SYS_EXTRACT_UTC(TIMESTAMP'2012-05-1719:15:26CET')
```

```
12/05/17 17:15:26,000000000
```

TO_TIMESTAMP()

Funkcja `TO_TIMESTAMP(x, format)` konwertuje napis *x* (który może być typu CHAR, VARCHAR2, NCHAR lub NVARCHAR2) na typ TIMESTAMP. Można również przesyłać opcjonalny parametr *format* dla *x*.

Poniższe zapytanie konwertuje na typ TIMESTAMP napis 2012-05-13 07:15:31.1234 w formacie YYYY-MM-DD HH24:MI:SS.FF:

```
SELECT TO_TIMESTAMP('2012-05-13 07:15:31.1234', 'YYYY-MM-DD HH24:MI:SS.FF')
FROM dual;
```

```
TO_TIMESTAMP('2012-05-1307:15:31.1234', 'YYYY-MM-DDHH24:MI:SS.FF')
```

```
-----  
12/05/13 07:15:31,123400000
```

TO_TIMESTAMP_TZ()

Funkcja `TO_TIMESTAMP_TZ(x, [format])` konwertuje *x* na typ TIMESTAMP WITH TIMEZONE. Można przesyłać opcjonalny parametr *format* dla *x*.

Poniższe zapytanie przesyła strefę czasową CET (identyfikowaną za pomocą TZR w napisie formatującym) do funkcji `TO_TIMESTAMP_TZ()`:

```
SELECT TO_TIMESTAMP_TZ('2012-05-13 07:15:31.1234 CET',
  'YYYY-MM-DD HH24:MI:SS.FF TZR')
FROM dual;
```

```
TO_TIMESTAMP_TZ('2012-05-1307:15:31.1234CET', 'YYYY-MM-DDHH24:MI:SS.FFTZR')
```

```
-----  
12/05/13 07:15:31,123400000 CET
```

Kolejne zapytanie wykorzystuje przesunięcie strefy czasowej o -7:00 względem UTC (-7:00 jest identyfikowane przez TZH i TZM w napisie formatującym):

```
SELECT TO_TIMESTAMP_TZ('2012-05-13 07:15:31.1234 -7:00',
  'YYYY-MM-DD HH24:MI:SS.FF TZH:TZM')
FROM dual;
```

```
TO_TIMESTAMP_TZ('2012-05-1307:15:31.1234-7:00', 'YYYY-MM-DDHH24:MI:SS.FFTZH:')
```

```
-----  
12/05/13 07:15:31,123400000 -07:00
```

Konwertowanie napisu na typ TIMESTAMP WITH LOCAL TIME ZONE

Za pomocą funkcji `CAST()` możemy konwertować napis na typ TIMESTAMP WITH LOCAL TIME ZONE.

Funkcja `CAST()` została przedstawiona w poprzednim rozdziale. Przypomnijmy, że `CAST(x AS typ)` konwertuje *x* na kompatybilny typ z bazy danych, określony przez parametr *typ*.

W poniższym zapytaniu wykorzystano funkcję `CAST()` do konwersji napisu 13-CZE-12 na typ TIMESTAMP WITH LOCAL TIME ZONE:

```
SELECT CAST('13-CZE-12' AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST('13-CZE-12'ASTIMESTAMPWITHLOCALTIMEZONE)
```

```
-----  
13/06/12 00:00:00,000000
```

Datownik zwrócony przez to zapytanie zawiera datę 13 czerwca 2012 roku i godzinę 0:00.

Kolejne zapytanie wykorzystuje funkcję `CAST()` do konwersji bardziej złożonego napisu na typ TIMESTAMP WITH LOCAL TIME ZONE:

```
SELECT CAST(TO_TIMESTAMP_TZ('2012-05-13 07:15:31.1234 CET',
  'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST(TO_TIMESTAMP_TZ('2012-05-1307:15:31.1234CET','YYYY-MM-DDHH24:MI:SS.FFT
-----12/05/13 07:15:31,123400
```

Znacznik czasu zwrócony przez to zapytanie zawiera datę 13 maja 2012 roku i godzinę 7:15:31,1234 CET (CET jest strefą czasową zarówno bazy danych, jak i sesji).

Poniższe zapytanie jest podobne do poprzedniego, z tym że tym razem zastosowano strefę czasową EST:

```
SELECT CAST(TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 EST',
    'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST(TO_TIMESTAMP_TZ('2012-05-1307:15:31.1234EST','YYYY-MM-DDHH24:MI:SS.FFT
-----12/05/13 14:15:31,123400
```

Datownik zwrócony przez to zapytanie zawiera datę 13 maja 2012 roku i godzinę 14:15:31,1234 (jako że czas w strefie EST jest przesunięty o siedem godzin do tyłu względem CET).

Interwały czasowe

W Oracle Database 9i wprowadzono możliwość składowania **interwałów czasowych**, których przykładami są:

- 1 rok i 3 miesiące,
- 25 miesięcy,
- –3 dni 5 godzin i 16 minut,
- 1 dzień i 7 godzin,
- –56 godzin.



Nie należy mylić interwałów czasowych z wyrażeniami *DataGodzina* i datownikami. Interwał czasowy rejestruje pewien okres (na przykład 1 rok i 3 miesiące), wyrażenie *DataGodzina* i datownik rejestrują natomiast konkretną datę i czas (na przykład 7:32:16 28 października 2006).

Załóżmy, że w naszym fikcyjnym sklepie chcemy wprowadzić okresowe rabaty, na przykład możemy dawać klientom kupony ważne przez kilka miesięcy lub po prostu przeprowadzić kilkudniową promocję.

Tabela 5.10 zawiera opis typów interwałów, a sposób ich użycia zostanie przedstawiony w kolejnych podrozdziałach.

Tabela 5.10. Typy interwałów czasowych

Typ	Opis
INTERVAL YEAR[(<i>precyzja_lat</i>)] TO MONTH	Składa się interwał czasowy wyrażony w latach i miesiącach. Można określić opcjonalną precyzję lat, przesyłając parametr <i>precyzja_lat</i> , który może być liczbą całkowitą od 0 do 9. Domyślną precyzją jest 2, co oznacza, że w interwale można przechować dwie cyfry roku. Jeżeli spróbujemy dodać wiersz zawierający więcej cyfr roku, niż może przechować kolumna INTERVAL YEAR TO MONTH, wystąpi błąd. Można składać ujemne i dodatnie interwały
INTERVAL DAY[(<i>precyzja_dni</i>)] TO SECOND [(<i>precyzja_sekund</i>)]	Składa się interwał czasowy wyrażony w dniach i sekundach. Można określić opcjonalną precyzję dni, przesyłając parametr <i>precyzja_dni</i> (liczbę całkowitą od 0 do 9, domyślnie 2). Można ponadto określić precyzję ułamków sekundy, przesyłając parametr <i>precyzja_sekund</i> (liczbę całkowitą od 0 do 9, domyślnie 6). Można składać dodatnie i ujemne interwały

Typ INTERVAL YEAR TO MONTH

Typ INTERVAL YEAR TO MONTH przechowuje interwał czasowy mierzony w latach i miesiącach. Poniższa instrukcja tworzy tabelę coupons, przechowującą informacje o kuponach. Tabela ta zawiera kolumnę typu INTERVAL YEAR TO MONTH o nazwie duration, w której będzie zapisywany okres ważności kuponu. Dla tej kolumny określono precyzję 3, co oznacza, że mogą być przechowywane maksymalnie trzy cyfry roku:

```
CREATE TABLE coupons (
    coupon_id INTEGER CONSTRAINT coupons_pk PRIMARY KEY,
    name VARCHAR2(32) NOT NULL,
    duration INTERVAL YEAR(3) TO MONTH
);
```

Do przesłania literała typu INTERVAL YEAR TO MONTH do bazy danych używamy uproszczonej składni:
INTERVAL '[+|-][y] [-m]' [YEAR[(precyzja_lat)]] [TO MONTH]

gdzie:

- + lub – jest opcjonalnym wskaźnikiem znaku interwału (domyślnie dodatnim),
- y jest opcjonalną liczbą lat w interwale,
- m jest opcjonalną liczbą miesięcy w interwale (jeżeli są przesyłane lata i miesiące, konieczne jest umieszczenie w literale TO MONTH),
- precyzja_lat jest opcjonalnym określeniem precyzji lat (domyślnie 2).

Tabela 5.11 zawiera przykładowe interwały typu YEAR TO MONTH.

Tabela 5.11. Przykładowe interwały typu YEAR TO MONTH

Literal	Opis
INTERVAL '1' YEAR	1 rok
INTERVAL '11' MONTH	11 miesięcy
INTERVAL '14' MONTH	14 miesięcy (czyli 1 rok i 2 miesiące)
INTERVAL '1-3' YEAR TO MONTH	1 rok i 3 miesiące
INTERVAL '0-5' YEAR TO MONTH	0 lat i 5 miesięcy
INTERVAL '123' YEAR(3) TO MONTH	123 lata przy precyzji trzech cyfr
INTERVAL '-5-1' YEAR TO MONTH	Ujemny interwał 1 rok i 3 miesiące
INTERVAL '1234' YEAR(3)	Nieprawidłowy interwał. 1234 zawiera cztery cyfry, a więc o jedną za dużo przy precyzji ustalonej na trzy cyfry

Poniższe instrukcje INSERT wstawiają do tabeli coupons wiersze, w których pole duration ma niektóre wartości spośród prezentowanych w powyższej tabeli:

```
INSERT INTO coupons (coupon_id, name, duration)
VALUES (1, 'Z File 1 zł taniej', INTERVAL '1' YEAR);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (2, 'Pop 3 2 zł taniej', INTERVAL '11' MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (3, 'Nauka współczesna 3 zł taniej', INTERVAL '14' MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (4, 'Wojny czołgów 2 zł taniej', INTERVAL '1-3' YEAR TO MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (5, 'Chemia 1 zł taniej', INTERVAL '0-5' YEAR TO MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (6, 'Twórczy wrzask 2 zł taniej', INTERVAL '123' YEAR(3));
```

Jeżeli spróbujemy dodać wiersz z kolumną duration ustawioną na nieprawidłowy interwał INTERVAL '1234' YEAR(3), wystąpi błąd, ponieważ precyza w kolumnie duration wynosi 3, więc jest niewystarczająca do przechowania liczby 1234. Poniższa instrukcja INSERT demonstruje wystąpienie tego błędu:

```
SQL> INSERT INTO coupons (coupon_id, name, duration)
  2 VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3));
VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3))
      *
```

BŁĄD w linii 2:

ORA-01873: wiodąca precyza interwału jest za mała

Poniższe zapytanie pobiera wiersze z tabeli coupons:

```
SELECT *
FROM coupons;
```

COUPON_ID	NAME	DURATION
1	Z Files 1 zł taniej	+001-00
2	Pop 2 zł taniej	+000-11
3	Nauka współczesna 3 zł taniej	+001-02
4	Wojny czołgów 2 zł taniej	+001-03
5	Chemia 1 zł taniej	+000-05
6	Twórczy wrzask 2 zł taniej	+123-00

Typ INTERVAL DAY TO SECOND

Typ INTERVAL DAY TO SECOND przechowuje interwały czasowe mierzone w dniach i sekundach. Poniższa instrukcja tworzy tabelę o nazwie promotions, w której będą przechowywane informacje o promocjach. Tabela ta zawiera kolumnę o nazwie duration o typie INTERVAL DAY TO SECOND, w której będą zapisywane okresy obowiązywania promocji:

```
CREATE TABLE promotions (
  promotion_id INTEGER CONSTRAINT promotions_pk PRIMARY KEY,
  name VARCHAR2(30) NOT NULL,
  duration INTERVAL DAY(3) TO SECOND (4)
);
```

Należy zauważyc, że w kolumnie duration dla dni określono precyzję 3, a dla ułamków sekund — 4. To oznacza, że w przypadku dnia mogą być przechowywane maksymalnie trzy cyfry, a w przypadku ułamkowych części sekundy — do czterech miejsc po przecinku.

Do przesłania literala typu INTERVAL DAY TO SECOND do bazy danych używamy uproszczonej składni:

```
INTERVAL '[+|‐][d] [:m[:s]]' [DAY[(precyza_dni)]]]
[TO HOUR | MINUTE | SECOND[(precyza_sekund)]]]
```

gdzie:

- + lub – jest opcjonalnym wskaźnikiem znaku interwału (domyślnie dodatnim),
- *d* jest liczbą dni w interwale,
- *h* jest opcjonalną liczbą godzin w interwale (jeżeli są przesyłane dni i godziny, konieczne jest umieszczenie w literale TO HOUR),
- *m* jest opcjonalną liczbą minut w interwale (jeżeli są przesyłane dni i minuty, konieczne jest umieszczenie w literale TO MINUTES),
- *s* jest opcjonalną liczbą sekund w interwale (jeżeli są przesyłane dni i sekundy, konieczne jest umieszczenie w literale TO SECOND),
- *precyza_dni* jest opcjonalnym określeniem precyzji dla dni (domyślnie 2),
- *precyza_sekund* jest opcjonalnym określeniem precyzji dla ułamków sekund (domyślnie 6).

Tabela 5.12 zawiera przykładowe interwały typu DAY TO SECOND.

Tabela 5.12. Przykładowe interwały typu DAY TO SECOND

Literal	Opis
INTERVAL '3' DAY	Interwał 3 dni
INTERVAL '2' HOUR	Interwał 2 godziny
INTERVAL '25' MINUTE	Interwał 25 minut
INTERVAL '45' SECOND	Interwał 45 sekund
INTERVAL '3 2' DAY TO HOUR	Interwał 3 dni i 2 godziny
INTERVAL '3 2:25' DAY TO MINUTE	Interwał 3 dni, 2 godziny i 25 minut
INTERVAL '3 2:25:45' DAY TO SECOND	Interwał 3 dni, 2 godziny, 25 minut i 45 sekund
INTERVAL '123 2:25:45.12' DAY(3) TO SECOND(2)	Interwał 123 dni, 25 minut, 45,12 sekund. Precyza dla dni wynosi 3, a dla ułamkowych części sekundy 2
INTERVAL '3 2:00:45' DAY TO SECOND	Interwał 3 dni, 2 godziny, 0 minut i 45 sekund
INTERVAL '-3 2:25:45' DAY TO SECOND	Ujemny interwał 3 dni, 2 godziny, 25 minut i 45 sekund
INTERVAL '1234 2:25:45' DAY(3) TO SECOND	Nieprawidłowy interwał, ponieważ liczba cyfr w dniu przekracza precyzję, określona jako 3
INTERVAL '123 2:25:45.123' DAY TO SECOND(2)	Nieprawidłowy interwał, ponieważ liczba cyfr w ułamkowej części sekund przekracza precyzję ustaloną jako 2

Poniższe instrukcje INSERT wstawiają wiersze do tabeli promotions:

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (1, '10% taniej Z Files', INTERVAL '3' DAY);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (2, '20% taniej Pop 3', INTERVAL '2' HOUR);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (3, '30% taniej Nauka współczesna', INTERVAL '25' MINUTE);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (4, '20% taniej Wojny czołgów', INTERVAL '45' SECOND);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (5, '10% taniej Chemia', INTERVAL '3 2:25' DAY TO MINUTE);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (6, '20% taniej Twórczy wrzask', INTERVAL '3 2:25:45' DAY TO SECOND);
```

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (7, '15% taniej Pierwsza linia', INTERVAL '123 2:25:45.12' DAY(3) TO SECOND(2));
```

Poniższe zapytanie pobiera wiersze z tabeli promotions:

```
SELECT *
FROM promotions;
```

PROMOTION_ID	NAME	DURATION
1	10% taniej Z Files	+003 00:00:00.0000
2	20% taniej Pop 3	+000 02:00:00.0000
3	30% taniej Nauka współczesna	+000 00:25:00.0000
4	20% taniej Wojny czołgów	+000 00:00:45.0000
5	10% taniej Chemia	+003 02:25:00.0000
6	20% taniej Twórczy wrzask	+003 02:25:45.0000
7	15% taniej Pierwsza linia	+123 02:25:45.1200

Funkcje operujące na interwałach

Mamy do dyspozycji kilka funkcji umożliwiających pobieranie i przetwarzanie interwałów czasowych. Zostały one przedstawione w tabeli 5.13, a ich szczegółowe omówienie znajduje się w kolejnych podrozdziałach.

Tabela 5.13. Funkcje operujące na interwałach

Funkcja	Opis
NUMTODSINTERVAL(x, jednostka_interwatu)	Konwertuje liczbę x na typ INTERVAL DAY TO SECOND. Interwał x jest określany przez parametr <i>jednostka_interwatu</i> , który może mieć jedną z następujących wartości: DAY, HOUR, MINUTE lub SECOND
NUMTOYMINTEGRAL(x, jednostka_interwatu)	Konwertuje liczbę x na typ INTERVAL YEAR TO MONTH. Interwał x jest określany przez parametr <i>jednostka_interwatu</i> , który może mieć wartość YEAR lub MONTH
TO_DSINTERVAL(x)	Konwertuje napis x na typ INTERVAL DAY TO SECOND
TO_YMINTERVAL(x)	Konwertuje napis x na typ INTERVAL YEAR TO MONTH

NUMTODSINTERVAL()

Funkcja NUMTODSINTERVAL(x, *jednostka_interwatu*) konwertuje liczbę x na typ INTERVAL DAY TO SECOND. Interwał dla x jest określany przez parametr *jednostka_interwatu*, który może mieć jedną z następujących wartości: DAY, HOUR, MINUTE lub SECOND.

Na przykład poniższe zapytanie konwertuje kilka liczb na interwały czasowe, korzystając z funkcji NUMTODSINTERVAL():

```
SELECT
  NUMTODSINTERVAL(1.5, 'DAY'),
  NUMTODSINTERVAL(3.25, 'HOUR'),
  NUMTODSINTERVAL(5, 'MINUTE'),
  NUMTODSINTERVAL(10.123456789, 'SECOND')
FROM dual;
```

```
NUMTODSINTERVAL(1.5, 'DAY')
-----
NUMTODSINTERVAL(3.25, 'HOUR')
-----
NUMTODSINTERVAL(5, 'MINUTE')
-----
NUMTODSINTERVAL(10.123456789, 'SECOND')
-----
+000000001 12:00:00.000000000
+000000000 03:15:00.000000000
+000000000 00:05:00.000000000
+000000000 00:00:10.123456789
```

NUMTOYMINTEGRAL()

Funkcja NUMTOYMINTEGRAL(x, *jednostka_interwatu*) konwertuje liczbę x na typ INTERVAL YEAR TO MONTH. Interwał dla x jest określany przez parametr *jednostka_interwatu*, który może mieć wartość YEAR lub MONTH.

Na przykład poniższe zapytanie konwertuje dwie liczby na interwały czasowe, korzystając z funkcji NUMTOYMINTEGRAL():

```
SELECT
  NUMTOYMINTEGRAL(1.5, 'YEAR'),
  NUMTOYMINTEGRAL(3.25, 'MONTH')
FROM dual;
```

```
NUMTOYMINTEGRAL(1.5, 'YEAR')
-----
```

```
NUMTOMINTERVAL(3.25, 'MONTH')
-----
+00000001-06
+00000000-03
```

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- typ `DATE` przechowuje wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minuty i sekundy,
- za pomocą typu `DATE` można składować wyrażenia *DataGodzina*,
- strefę czasową określamy, podając przesunięcie względem UTC albo za pomocą nazwy regionu (na przykład CET),
- datownik przechowuje wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), minuty, sekundy, ułamki sekund i strefę czasową,
- interwał przechowuje informację o długości odcinka czasu. Przykładem interwału czasowego jest rok i trzy miesiące.

W kolejnym rozdziale zawarto informacje o tym, jak zagnieździć jedno zapytanie w innym.

ROZDZIAŁ

6

Podzapytania

W tym rozdziale będzie mowa o:

- umieszczaniu wewnętrznej instrukcji SELECT w zewnętrznej instrukcji SELECT, UPDATE lub DELETE. Wewnętrzną instrukcję SELECT nazywamy **podzapytaniem**,
- sposobach użycia różnych rodzajów podzapytań,
- tym, jak podzapytania pozwalają budować bardzo złożone instrukcje z prostych elementów.

Rodzaje podzapytań

Wyróżniamy dwa podstawowe rodzaje podzapytań:

- **Podzapytania jednowierszowe** — zwracają do zewnętrznej instrukcji SQL zero lub jeden wiersz. Istnieje specjalny przypadek jednowierszowego podzapytania, który zawiera dokładnie jedną kolumnę. Tego rodzaju podzapytanie nazywamy **podzapytaniem skalarnym**.
- **Podzapytania wielowierszowe** — zwracają do zewnętrznej instrukcji SQL co najmniej jeden wiersz. Wyróżniamy ponadto trzy rodzaje podzapytań, które mogą zwrócić jeden wiersz lub wiele wierszy:
 - **Podzapytania wielokolumnowe** — zwracają do zewnętrznej instrukcji SQL więcej niż jedną kolumnę.
 - **Podzapytania skorelowane** — odwołują się do jednej lub kilku kolumn z zewnętrznej instrukcji SQL. Takie podzapytania nazywamy „skorelowanymi”, ponieważ są one powiązane z zewnętrzna instrukcją SQL poprzez te same kolumny.
 - **Podzapytania zagnieżdzone** — są umieszczane wewnętrz innego podzapytania.

W tym rozdziale zostaną opisane wszystkie wymienione rodzaje podzapytań oraz sposoby dodawania ich do instrukcji SELECT, UPDATE i DELETE.

Pisanie podzapytań jednowierszowych

Podzapytanie jednowierszowe zwraca do zewnętrznej instrukcji SQL zero lub jeden wiersz. Podzapytanie możemy umieścić w klauzuli WHERE, klauzuli HAVING lub klauzuli FROM instrukcji SELECT.

Podzapytania w klauzuli WHERE

Podzapytanie można umieścić w klauzuli WHERE innego zapytania. Klauzula WHERE poniższego zapytania zawiera podzapytanie. Należy zauważać, że zostało ono umieszczone w nawiasach okrągłych ():

```
SELECT first_name, last_name
FROM customers
WHERE customer_id =
  (SELECT customer_id
   FROM customers
   WHERE last_name = 'Nikiel');
```

FIRST_NAME	LAST_NAME
Jan	Nikiel

Ten przykład pobiera wartości pól `first_name` i `last_name` z wiersza tabeli `customers`, w którym `last_name` ma wartość Nikiel.

Podzapytanie w klauzuli `WHERE` ma postać:

```
SELECT customer_id
FROM customers
WHERE last_name = 'Nikiel';
```

To zapytanie jest uruchamiane jako pierwsze (i tylko jeden raz) i zwraca `customer_id` dla wiersza, w którym pole `last_name` ma wartość Nikiel. Wartość `customer_id` w tym wierszu wynosi 1 i jest ona przesyłana do klauzuli `WHERE` zewnętrznego zapytania, dlatego też zwraca ono ten sam wynik co poniższe:

```
SELECT first_name, last_name
FROM customers
WHERE customer_id = 1;
```

Użycie innych operatorów jednowierszowych

W klauzuli `WHERE` podzapytania przedstawionego na początku poprzedniego podrozdziału był wykorzystywany operator równości (=). W podzapytaniach jednowierszowych można również używać innych operatorów porównania, takich jak `<`, `>`, `<=` i `>=`.

W poniższym przykładzie w klauzuli `WHERE` zewnętrznego zapytania użyto operatora `>`. W podzapytaniu zastosowano funkcję `AVG()` do obliczenia średniej ceny produktów, która jest przesyłana do klauzuli `WHERE` zewnętrznego zapytania. Całe zapytanie zwraca `product_id`, `name` i `price` towarów, których cena jest większa od średniej ceny wszystkich produktów:

```
SELECT product_id, name, price
FROM products
WHERE price >
  (SELECT AVG(price)
   FROM products);
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	19,95
2	Chemia	30
3	Supernowa	25,99
5	Z Files	49,99

Rozbijmy to zapytanie, aby zrozumieć, jak działa. Poniższy przykład przedstawia uruchomienie samego podzapytania:

```
SELECT AVG(price)
FROM products;
```

AVG(PRICE)
19,7308333

To podzapytanie jest przykładem podzapytania skalarnego, ponieważ zwraca dokładnie jeden wiersz zawierający jedną kolumnę. Wartość zwracaną przez podzapytanie skalarne jest traktowana jako jedna wartość skalarna.



Uwaga

Wartość 19,7308333 zwrócona przez podzapytanie jest wykorzystywana w klauzuli WHERE zewnętrznego zapytania, w związku z czym jest ono tożsame z poniższym:

```
SELECT product_id, name, price
FROM products
WHERE price > 19.7308333;
```

Podzapytania w klauzuli HAVING

Jak widzieliśmy w rozdziale 4., klauzula HAVING służy do filtrowania grup wierszy. W klauzuli HAVING zewnętrznego zapytania możemy umieścić podzapytanie. To pozwala nam filtrować grupy wierszy na podstawie wyników zwracanych przez podzapytanie.

W poniższym przykładzie użyto podzapytanego w klauzuli HAVING zewnętrznego zapytania. Przykład pobiera identyfikator (product_type_id) oraz średnią cenę produktów danego typu, jeżeli jest ona mniejsza niż największa spośród średnich cen wszystkich typów produktów:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
       (SELECT MAX(AVG(price))
        FROM products
        GROUP BY product_type_id)
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	Avg(PRICE)
1	24,975
3	13,24
4	13,99
	13,49

Zauważmy, że podzapytanie oblicza najpierw średnią cenę każdego rodzaju produktów za pomocą funkcji AVG(). Wynik zwrócony przez tę funkcję jest przesyłany do funkcji MAX(), która zwraca największą średnią cenę.

Przeanalizujmy przykład, aby zrozumieć działanie tego zapytania. Oto wynik uruchomienia samego podzapytania:

```
SELECT MAX(AVG(price))
FROM products
GROUP BY product_type_id;

MAX(AVG(PRICE))
-----
26,22
```

Wartość 26,22 jest wykorzystywana w klauzuli HAVING zewnętrznego zapytania do odfiltrowania tylko tych wierszy grup, w których średnia cena jest mniejsza od 26,22. Poniższe zapytanie przedstawia wersję zewnętrznego zapytania, która pobiera product_type_id oraz średnią cenę produktów pogrupowanych według product_type_id:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;

PRODUCT_TYPE_ID AVG(PRICE)
-----
1      24,975
2      26,22
3      13,24
4      13,99
      13,49
```

Średnia cena mniejsza niż 26,22 występuje w grupach z `product_type_id` równym 1, 3, 4 i `null`. Zgodnie z oczekiwaniami te same grupy zostały zwrocone przez zapytanie przedstawione na początku tego podrozdziału.

Podzapytania w klauzuli FROM (widoki wbudowane)

Podzapytanie można umieścić w klauzuli FROM zewnętrznego zapytania. Tego typu podzapytania nazywamy **widokami wbudowanymi** (ang. *inline view*).

Poniższy, prosty przykład pobiera produkty, dla których `product_id` jest mniejsze od 3:

```
SELECT product_id
FROM
  (SELECT product_id
   FROM products
   WHERE product_id < 3);
```

PRODUCT_ID
1
2

Podzapytanie zwraca do zapytania zewnętrznego wiersze z tabeli `products`, w których `product_id` ma wartość mniejszą od 3. Zapytanie zewnętrzne pobiera i wyświetla te wartości `product_id`. Z perspektywy klauzuli FROM zewnętrznego zapytania wyniki podzapytania są po prostu kolejnym źródłem danych.

Kolejny przykład jest bardziej użyteczny i w zewnętrznym zapytaniu pobiera wartości `product_id` oraz `price` z tabeli `products`, a podzapytanie pobiera liczbę dokonanych zakupów danego produktu:

```
SELECT prds.product_id, price, purchases_data.product_count
FROM products prds,
  (SELECT product_id, COUNT(product_id) product_count
   FROM purchases
   GROUP BY product_id) purchases_data
WHERE prds.product_id = purchases_data.product_id;
```

PRODUCT_ID	PRICE	PRODUCT_COUNT
1	19,95	4
2	30	4
3	25,99	1

Należy zauważyć, że podzapytanie pobiera z tabeli `purchases` wartości `product_id` i `COUNT(product_id)` i zwraca je do zewnętrznego zapytania. Jak widzimy, wyniki z podzapytania są po prostu kolejnym źródłem danych dla klauzuli FROM zewnętrznego zapytania.

Błędy, które można napotkać

W tym podrozdziale opiszę kilka częstych błędów. Zobaczmy, że zapytanie jednowierszowe może zwrócić maksymalnie jeden wiersz oraz że tego typu zapytanie nie może zawierać klauzuli ORDER BY.

Zapytanie jednowierszowe może zwracać maksymalnie jeden wiersz

Jeżeli podzapytanie zwraca więcej niż jeden wiersz, zostanie wyświetlony poniższy komunikat o błędzie:

ORA-01427: jednowierszowe podzapytanie zwraca więcej niż jeden wiersz

Na przykład podzapytanie w poniższej instrukcji próbuje przesyłać kilka wierszy do operatora równości (=) w zewnętrznym zapytaniu:

```
SQL> SELECT product_id, name
  2  FROM products
  3 WHERE product_id =
  4   (SELECT product_id
    FROM products
  5   )
```

```
6 WHERE name LIKE '%e%');
```

```
(SELECT product_id
*
```

BŁĄD w linii 4:
ORA-01427: jednowierszowe podzapytanie zwraca więcej niż jeden wiersz

W tabeli products znajduje się siedem wierszy, w których nazwa produktu zawiera literę e i podzapytanie próbuje przesłać je do operatora równości w zewnętrznym zapytaniu. Ponieważ operator równości może obsługiwać tylko jeden wiersz, zapytanie jest nieprawidłowe i zwracany jest komunikat o błędzie.

Z podrozdziału „Pisanie podzapytań wielowierszowych” dowiesz się, jak zwrócić wiele wierszy z podzapytania.

Podzapytania nie mogą zawierać klauzuli ORDER BY

Podzapytanie nie może zawierać klauzuli ORDER BY. Zamiast tego sortowanie musi być wykonane przez zewnętrzne zapytanie. Na przykład poniższe zapytanie zewnętrzne zawiera na końcu klauzulę ORDER BY, sortującą wartości product_id w kolejności malejącej:

```
SELECT product_id, name, price
FROM products
WHERE price >
  (SELECT AVG(price)
   FROM products)
ORDER BY product_id DESC;
```

PRODUCT_ID	NAME	PRICE
5	Z Files	49,99
3	Supernowa	25,99
2	Chemia	30
1	Nauka współczesna	19,95

Pisanie podzapytań wielowierszowych

Podzapytania wielowierszowe zwracają jeden wiersz lub kilka wierszy do zapytania zewnętrznego. Zapytanie zewnętrzne może obsługiwać podzapytania zwracające wiele wierszy za pomocą operatorów IN, ANY lub ALL. Możemy użyć tych operatorów do sprawdzenia, czy wartość z kolumny znajduje się na liście wartości. Na przykład:

```
SELECT product_id, name
FROM products
WHERE product_id IN (1, 2, 3);
```

PRODUCT_ID	NAME
1	Nauka współczesna
2	Chemia
3	Supernowa

W tym rozdziale zobaczysz, że lista wartości może pochodzić z podzapytania.

 **Uwaga** Do sprawdzenia, czy wartość znajduje się na liście pochodzącej z podzapytania skorelowanego, możemy również użyć operatora EXISTS. Więcej informacji na ten temat znajduje się w podrozdziale „Pisanie podzapytań skorelowanych”.

Użycie operatora IN z podzapytaniem wielowierszowym

Operator IN pozwala sprawdzić, czy wartość znajduje się na konkretnej liście wartości. Ta lista może być wynikiem zwróconym przez podzapytanie. Możemy również wykonać logiczne zaprzeczenie IN, stosując operator NOT IN, co pozwala sprawdzić, czy wartość nie znajduje się na określonej liście wartości.

W poniższym przykładzie użyto operatora IN do sprawdzenia, czy `product_id` znajduje się na liście wartości zwróconej przez podzapytanie. Zwraca ono wartości `product_id` produktów, których nazwa zawiera literę e:

```
SELECT product_id, name
FROM products
WHERE product_id IN
  (SELECT product_id
   FROM products
   WHERE name LIKE '%e%');
```

PRODUCT_ID	NAME
1	Nauka współczesna
2	Chemia
3	Supernowa
5	Z Files
7	Space Force 9
8	Z innej planety
12	Pierwsza linia

W kolejnym przykładzie użyto operatora NOT IN, aby pobrać dane o produktach, które nie znajdują się w tabeli purchases:

```
SELECT product_id, name
FROM products
WHERE product_id NOT IN
  (SELECT product_id
   FROM purchases);
ORDER BY product_id;
```

PRODUCT_ID	NAME
4	Wojny czołgów
5	Z Files
6	2412: Powrót
7	Space Force 9
8	Z innej planety
9	Muzyka klasyczna
10	Pop 3
11	Twórczy wrzask
12	Pierwsza linia

Użycie operatora ANY z podzapytaniem wielowierszowym

Operator ANY służy do porównywania wartości z każdą z wartości na liście. Należy przed nim umieścić operator porównania =, <>, <, >, <= lub >=.

W poniższym przykładzie użyto operatora ANY do pobrania nazwisk pracowników, których wynagrodzenie jest niższe od najniższego wynagrodzenia w tabeli `salary_grades`:

```
SELECT employee_id, last_name
FROM employees
WHERE salary < ANY
  (SELECT low_salary
   FROM salary_grades);
ORDER BY employee_id;
```

EMPLOYEE_ID	LAST_NAME
2	Joświerz
3	Helc
4	Nowak

Użycie operatora ALL z podzapytaniem wielowierszowym

Operator ALL służy do porównywania wartości z każdą z wartości z listy. Należy przed nim umieścić operator porównania =, <, >, <= lub >=. W poniższym przykładzie użyto operatora ALL do pobrania nazwisk pracowników, których wynagrodzenie jest wyższe od najwyższego wynagrodzenia w tabeli salary_grades:

```
SELECT employee_id, last_name
FROM employees
WHERE salary > ALL
  (SELECT high_salary
   FROM salary_grades);
```

no rows selected

Jak widać, żaden pracownik nie otrzymuje wyższego wynagrodzenia niż najwyższe.

Pisanie podzapytań wielokolumnowych

Podzapytania mogą zwracać wiele kolumn. Poniższy przykład pobiera z każdej grupy produkty w najniższej cenie:

```
SELECT product_id, product_type_id, name, price
FROM products
WHERE (product_type_id, price) IN
  (SELECT product_type_id, MIN(price)
   FROM products
   GROUP BY product_type_id)
ORDER BY product_id;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Nauka współczesna	19,95
4	2	Wojny czołgów	13,95
8	3	Z innej planety	12,99
9	4	Muzyka klasyczna	10,99

Podzapytanie zwraca product_type_id oraz minimalną cenę (price) dla każdej grupy towarów i te wartości są porównywane w klauzuli WHERE zapytania zewnętrznego z product_type_id oraz ceną każdego produktu. Dla każdej grupy wyświetlany jest produkt z najniższą ceną.

Pisanie podzapytań skorelowanych

Podzapytanie skorelowane odwołuje się do jednej lub kilku kolumn z zewnętrznej instrukcji SQL. Takie zapytania nazywamy **skoreowanymi**, ponieważ są związane z zewnętrzną instrukcją SQL za pośrednictwem tych samych kolumn.

Podzapytania skorelowane są zwykle używane, gdy chcemy uzyskać odpowiedź na pytanie dotyczące wartości w każdym wierszu znajdującym się w zewnętrznym zapytaniu. Na przykład możemy chcieć sprawdzić, czy występuje relacja między danymi, ale nie interesuje nas, ile wierszy zostało zwróconych przez podzapytanie. To znaczy, że chcemy jedynie sprawdzić, czy zostały zwrócone *jakiekolwiek* wiersze.

Podzapytanie skorelowane jest wykonywane raz dla każdego wiersza w zapytaniu zewnętrznym, co odróżnia je od podzapytania nieskoreowanego, które jest uruchamiane jeden raz, przed uruchomieniem zapytania zewnętrznego. W kolejnych podrozdziałach znajdują się przykłady takich zapytań.

Przykład podzapytania skoreowanego

Poniższe podzapytanie skorelowane pobiera produkty, których cena jest wyższa niż średnia w grupie danego produktu:

```
SELECT product_id, product_type_id, name, price
FROM products outer
WHERE price >
  (SELECT AVG(price)
   FROM products inner
   WHERE inner.product_type_id = outer.product_type_id);
ORDER BY product_id;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
2	1	Chemia	30
5	2	Z Files	49,99
7	3	Space Force 9	13,49
10	4	Pop 3	15,99
11	4	Twórczy wrzask	14,99

Należy zauważyć użycie aliasu `outer` do oznaczenia zewnętrznego zapytania i aliasu `inner` do oznaczenia podzapytania. Odwołanie do kolumny `product_type_id` w części zewnętrznej i wewnętrznej sprawia, że wewnętrzne podzapytanie jest skorelowane z zapytaniem zewnętrznym. Poza tym podzapytanie zwraca jeden wiersz zawierający średnią cenę produktu.

W podzapytaniu skorelowanym każdy wiersz z zapytania zewnętrznego jest przesyłany po kolejno do podzapytania. Podzapytanie odczytuje i przetwarza każdy wiersz z zapytania zewnętrznego. Następnie zwracane są wyniki całego zapytania.

W powyższym przykładzie zewnętrzne zapytanie pobiera wszystkie wiersze z tabeli `products` i przesyła je do zapytania wewnętrznego. Każdy wiersz jest odczytywany przez zapytanie wewnętrzne. Oblicza ono średnią cenę każdego produktu, dla którego `product_type_id` w zapytaniu wewnętrznym jest równy `product_type_id` w zapytaniu zewnętrznym.

Użycie operatorów EXISTS i NOT EXISTS z podzapytaniem skorelowanym

Operator `EXISTS` służy do sprawdzenia, czy podzapytanie zwróciło jakiekolwiek wiersze. Choć można korzystać z tego operatora z podzapytaniami nieskoreowanymi, zwykle używamy go jednak z podzapytaniami skoreowanymi.

Operator `NOT EXISTS` stanowi logiczne zaprzeczenie `EXISTS`: sprawdza, czy podzapytanie nie zwróciło żadnych wierszy.

Użycie operatora EXISTS z podzapytaniem skorelowanym

W poniższym przykładzie użyto operatora `EXISTS` do pobrania nazwisk pracowników będących przełożonymi innych zatrudnionych. Należy zauważyć, że nie interesuje nas liczba wierszy zwróconych przez podzapytanie. Chcemy się jedynie dowiedzieć, czy w ogóle zostały zwrócone jakieś wiersze:

```
SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
  (SELECT employee_id
   FROM employees inner
   WHERE inner.manager_id = outer.employee_id)
ORDER BY employee_id;
```

EMPLOYEE_ID	LAST_NAME
1	Kowalski
2	Joświerz

Ponieważ operator `EXISTS` jedynie sprawdza, czy podzapytanie zwróciło jakieś wiersze, nie musi ono zwracać kolumny — może po prostu zwracać literał. Ta właściwość może poprawić jego wydajność. Zmieniono powyższe zapytanie tak, aby podzapytanie zwracało literał 1:

```
SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
  (SELECT 1
   FROM employees inner
   WHERE inner.manager_id = outer.employee_id);
ORDER BY employee_id;
```

```
EMPLOYEE_ID LAST_NAME
----- 
1 Kowalski
2 Joświerz
```

Jeżeli podzapytanie zwraca choć jeden wiersz, EXISTS zwraca wartość prawda. Jeżeli podzapytanie nie zwraca żadnych wierszy, EXISTS zwraca fałsz. W powyższych przykładach nie interesowała nas liczba wierszy zwracanych przez podzapytanie. Chcieliśmy się jedynie dowiedzieć, czy w ogóle zostały zwrócone jakieś wiersze, więc użyliśmy operatora EXISTS zwracającego wartość prawda lub fałsz. Ponieważ zapytanie zewnętrzne wymaga przynajmniej jednej kolumny, w powyższym przykładzie podzapytanie zwraca literał 1.

Użycie operatora NOT EXISTS w podzapytaniu skorelowanym

W poniższym przykładzie użyto operatora NOT EXISTS do pobrania produktów, które nie zostały kupione:

```
SELECT product_id, name
FROM products outer
WHERE NOT EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id)
ORDER BY product_id;
```

```
PRODUCT_ID NAME
----- 
4 Wojny czołgów
5 Z Files
6 2412: Powrót
7 Space Force 9
8 Z innej planety
9 Muzyka klasyczna
10 Pop 3
11 Twórczy wrzask
12 Pierwsza linia
```

Operatory EXISTS i NOT EXISTS a operatory IN i NOT IN

Operator IN sprawdza, czy wartość znajduje się na liście wartości. Operator EXISTS różni się od operatora IN, ponieważ sprawdza on jedynie istnienie wierszy, a IN sprawdza faktyczne wartości.

 Wskazówka Operator EXISTS w podzapytaniach oferuje zwykle większą wydajność niż IN, dlatego jeżeli jest to możliwe, należy używać operatora EXISTS zamiast IN.

Pisząc zapytania zawierające operator NOT EXISTS lub NOT IN, należy zachować ostrożność. Jeżeli lista wartości zawiera wartość NULL, operator NOT EXISTS zwróci wartość prawda, a NOT IN — fałsz. Rozważmy poniższy przykład, w którym wykorzystano operator NOT EXISTS do pobrania typów produktów, których egzemplarze nie istnieją w tabeli products:

```
SELECT product_type_id, name
FROM product_types outer
WHERE NOT EXISTS
  (SELECT 1
   FROM products inner
   WHERE inner.product_type_id = outer.product_type_id)
```

```
ORDER BY product_type_id;
```

```
PRODUCT_TYPE_ID NAME
```

```
-----  
5 Czasopismo
```

Został zwrócony jeden wiersz.

W kolejnym przykładzie zmodyfikowano poprzednie zapytanie tak, aby użyć operatora NOT IN. Nie zostały zwrócone żadne wiersze:

```
SELECT product_type_id, name  
FROM product_types  
WHERE product_type_id NOT IN  
(SELECT product_type_id  
FROM products);
```

```
no rows selected
```

Żadne wiersze nie zostały zwrócone, ponieważ podzapytanie zwraca listę wartości product_type_id, z których jedna to NULL (product_type_id produktu nr 12 wynosi NULL). Operator NOT IN w zapytaniu zewnętrznym zwraca więc wartość fałsz i nie są zwracane żadne wiersze. Ten problem można rozwiązać, używając funkcji NVL() do konwersji wartości NULL na inną. W poniższym przykładzie użyto tej funkcji do konwersji wartości NULL z kolumny product_type_id na 0:

```
SELECT product_type_id, name  
FROM product_types  
WHERE product_type_id NOT IN  
(SELECT NVL(product_type_id, 0)  
FROM products);
```

```
PRODUCT_TYPE_ID NAME
```

```
-----  
5 Czasopismo
```

Tym razem wiersz został pobrany.



Te przykłady ilustrują kolejną różnicę między podzapytaniami skorelowanymi i nieskorelowanymi — zapytanie skorelowanie potrafi obsługiwać wartość NULL.

Pisanie zagnieżdżonych podzapytań

Podzapytania można zagnieżdżać w innych podzapytaniach. Poniższy przykład zawiera zagnieżdżone podzapytanie. Należy zauważyć, że jest ono umieszczone w podzapytaniu, które z kolei jest umieszczone w zapytaniu zewnętrznym:

```
SELECT product_type_id, AVG(price)  
FROM products  
GROUP BY product_type_id  
HAVING AVG(price) <  
(SELECT MAX(AVG(price))  
FROM products  
WHERE product_type_id IN  
(SELECT product_id  
FROM purchases  
WHERE quantity > 1)  
GROUP BY product_type_id)  
ORDER BY product_type_id;
```

```
PRODUCT_TYPE_ID AVG(PRICE)
```

```
-----  
1      24,975  
3      13,24  
4      13,99  
      13,49
```

Ten przykład jest stosunkowo złożony i zawiera trzy zapytania: zagnieżdżone podzapytanie, podzapytanie i zapytanie zewnętrzne. Elementy te są wykonywane w takiej kolejności. Rozbijmy ten przykład na trzy części i przyjrzyjmy się zwracanym wynikom. Podzapytanie zagnieżdżone ma postać:

```
SELECT product_id
  FROM purchases
 WHERE quantity > 1
```

Zwraca ono `product_id` dla produktów, które zostały kupione więcej niż raz:

```
PRODUCT_ID
```

```
-----
 2
 1
```

Podzapytanie odbierające te wyniki ma postać:

```
SELECT MAX(AVG(price))
  FROM products
 WHERE product_type_id IN
 (... wyjście z zagnieżdżonego podzapytania ...)
 GROUP BY product_type_id
```

To podzapytanie zwraca maksymalną średnią cenę produktów zwróconych przez podzapytanie zagnieżdżone:

```
MAX(AVG(PRICE))
-----
 26,22
```

Ten wiersz jest zwracany do następującego zapytania zewnętrznego:

```
SELECT product_type_id, AVG(price)
  FROM products
 GROUP BY product_type_id
 HAVING AVG(price) <
 (... wyjście z podzapytania ...)
 ORDER BY product_type_id;
```

To zapytanie zwraca `product_type_id` oraz średnią cenę produktów, jeżeli jest ona niższa od średniej ceny zwróconej przez podzapytanie:

```
PRODUCT_TYPE_ID AVG(PRICE)
-----
 1      24,975
 3      13,24
 4      13,99
          13,49
```

Są to wiersze zwracane przez całe zapytanie, przedstawione na początku tego podrozdziału.

Pisanie instrukcji UPDATE i DELETE zawierających podzapytania

Podzapytania mogą być umieszczane w instrukcjach UPDATE i DELETE.

Pisanie instrukcji UPDATE zawierającej podzapytanie

W instrukcji UPDATE możemy przypisać kolumnie wartość będącą wynikiem zwróconym przez podzapytanie jednowierszowe. Na przykład poniższa instrukcja UPDATE przypisuje pracownikowi nr 4 wynagrodzenie będące średnią górnych granic przedziałów wynagrodzeń, która jest zwracana przez podzapytanie:

```
UPDATE employees
SET salary =
  (SELECT AVG(high_salary)
```

```
FROM salary_grades)
WHERE employee_id = 4;
```

1 row updated.

Wykonanie tej instrukcji zwiększa wynagrodzenie pracownika nr 4 z 500 000 do 625 000 zł (jest to średnia górnych granic przedziałów wynagrodzeń w tabeli salary_grades).

Kolejna instrukcja wycofuje zmiany wprowadzone do bazy:

ROLLBACK;

 **Uwaga** Po uruchomieniu instrukcji UPDATE należy wykonać instrukcję ROLLBACK, aby wycofać zmiany. Dzięki temu wyniki zwarcane przez zapytania będą zgodne z prezentowanymi w książce.

Pisanie instrukcji DELETE zawierającej podzapytanie

Wiersze zwarcane przez podzapytanie możemy wykorzystać w klauzuli WHERE instrukcji DELETE. Na przykład poniższa instrukcja DELETE usuwa dane pracownika, którego wynagrodzenie jest większe niż średnia górnych granic przedziałów wynagrodzeń zwarcana przez podzapytanie:

```
DELETE FROM employees
WHERE salary >
  (SELECT AVG(high_salary)
   FROM salary_grades);
```

1 row deleted.

Ta instrukcja DELETE usuwa dane pracownika nr 1.

Kolejna instrukcja wycofuje zmiany wprowadzone do bazy:

ROLLBACK;

 **Uwaga** Po wykonaniu instrukcji DELETE należy pamiętać o uruchomieniu instrukcji ROLLBACK, aby wycofać usunięcie wiersza.

Przygotowywanie podzapytań

Wewnątrz klauzuli WITH można umieszczać podzapytania, do których odwoływać się można poza klauzulą WITH. Można to nazwać przygotowywaniem podzapytań (ang. *subquery factoring*).

Poniższy przykład zawiera wewnątrz klauzuli WITH podzapytanie nazwane customer_purchases. Główne podzapytanie na zewnątrz klauzuli WITH zwraca wyniki z podzapytania customer_purchases.

```
WITH
  customer_purchases AS (
    SELECT
      cu.customer_id,
      SUM(pr.price * pu.quantity) AS purchase_total
    FROM customers cu, purchases pu, products pr
    WHERE cu.customer_id = pu.customer_id
    AND pu.product_id = pr.product_id
    GROUP BY cu.customer_id
  )
SELECT *
FROM customer_purchases
ORDER BY customer_id;
```

CUSTOMER_ID	PURCHASE_TOTAL
1	109,95
2	69,9
3	75,94
4	49,95

Średnia sumarycznych cen zakupu wynosi 76,435.

Następny przykład rozszerza poprzednie zapytanie. Podzapytanie `customer_purchases` pobiera identyfikatory klientów, sumaryczną cenę ich zakupów oraz średnią cenę poszczególnych zakupów. Główne zapytanie zawierające klauzulę `WITH` zwraca identyfikator klienta oraz sumaryczną kwotę zakupów klientów, w przypadku których sumaryczna kwota zakupów jest mniejsza niż średnia kwota wszystkich zakupów.

```
WITH
  customer_purchases AS (
    SELECT
      cu.customer_id,
      SUM(pr.price * pu.quantity) AS purchase_total
    FROM customers cu, purchases pu, products pr
    WHERE cu.customer_id = pu.customer_id
    AND pu.product_id = pr.product_id
    GROUP BY cu.customer_id
  ),
  average_purchase AS (
    SELECT SUM(purchase_total)/COUNT(*) AS average
    FROM customer_purchases
  )
SELECT *
FROM customer_purchases
WHERE purchase_total < (
  SELECT average
  FROM average_purchase
)
ORDER BY customer_id;

CUSTOMER_ID PURCHASE_TOTAL
-----
 2          69,9
 3          75,94
 4          49,95
```

Średnia sumarycznych kwot zakupów poszczególnych klientów to 76,435, dlatego w wynikach zwróconych przez przykład znajdują się informacje o klientach, dla których suma zakupów jest mniejsza niż 76,435.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- podzapytanie jest zapytaniem umieszczonym w instrukcji `SELECT`, `UPDATE` lub `DELETE`,
- podzapytania jednowierszowe zwracają zero wierszy lub jeden wiersz,
- podzapytania wielowierszowe zwracają co najmniej jeden wiersz,
- podzapytania wielokolumnowe zwracają więcej niż jedną kolumnę,
- podzapytania skorelowane odwołują się do jednej kolumny lub kilku z zewnętrznej instrukcji SQL,
- podzapytania zagnieżdżone są podzapytaniami umieszczonymi w innych podzapytaniach,
- podzapytania mogą być też umieszczone wewnątrz klauzuli `WITH`.

W kolejnym rozdziale zawarto informacje o bardziej złożonych zapytaniach.

ROZDZIAŁ

7

Zapytania zaawansowane

Z tego rozdziału dowiesz się, jak:

- używać operatorów zestawu, które umożliwiają łączenie wyników zwracanych przez dwa zapytania lub więcej,
- używać funkcji `TRANSLATE()` do zastępowania znaków w jednym napisie znakami z innego napisu,
- używać funkcji `DECODE()` do wyszukiwania określonej wartości w zestawie wartości,
- używać wyrażenia `CASE` do wykonywania logiki `if-then-else` w SQL,
- wykonywać zapytania o dane **hierarchiczne**,
- używać klauzul `ROLLUP i CUBE` do obliczania sum pośrednich i całkowitych grup wierszy,
- łączyć wiersze z dwóch instrukcji `SELECT` za pomocą `CROSS APPLY i OUTER APPLY`,
- zwrócić wbudowany widok danych za pomocą instrukcji `LATERAL`.

Operatory zestawu

Operatory zestawu umożliwiają łączenie wierszy zwracanych przez dwa zapytania lub więcej. W tabeli 7.1 przedstawiono cztery operatory zestawu.

Tabela 7.1. Operatory zestawu

Operator	Opis
UNION ALL	Zwraca wszystkie wiersze pobrane przez zapytania, łącznie z wierszami powtarzającymi się
UNION	Zwraca wszystkie niepowtarzające się wiersze pobrane przez zapytania
INTERSECT	Zwraca wiersze, które zostały pobrane przez obydwa zapytania
MINUS	Zwraca wiersze pozostałe po odjęciu wierszy pobranych przez drugie zapytanie od wierszy pobranych przez pierwsze zapytanie

Gdy korzysta się z operatorów zestawu, należy pamiętać o następującym ograniczeniu: *liczba kolumn i typy kolumn zwracane przez zapytania muszą być takie same, nazwy kolumn natomiast mogą się różnić*.

Wkrótce dowiesz się, jak stosować operatory przedstawione w tabeli 7.1, ale zaczniemy od przyjrzenia się przykładowym tabelom wykorzystywanym w tym podrozdziale.

Przykładowe tabele

Tabele `products` i `more_products` są tworzone przez skrypt `store_schema.sql` za pomocą następujących instrukcji:

```

CREATE TABLE products (
    product_id INTEGER
        CONSTRAINT products_pk PRIMARY KEY,
    product_type_id INTEGER
        CONSTRAINT products_fk_product_types
            REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);
CREATE TABLE more_products (
    prd_id INTEGER
        CONSTRAINT more_products_pk PRIMARY KEY,
    prd_type_id INTEGER
        CONSTRAINT more_products_fk_product_types
            REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    available CHAR(1)
);

```

Poniższe zapytanie pobiera kolumny `product_id`, `product_type_id` oraz `name` z tabeli `products`:

```

SELECT product_id, product_type_id, name
FROM products;

```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Chemia
3	2	Supernowa
4	2	Wojny czołgów
5	2	Z Files
6	2	2412: Powrót
7	3	Space Force 9
8	3	Z innej planety
9	4	Muzyka klasyczna
10	4	Pop 3
11	4	Twórczy wrzask
12		Pierwsza linia

Kolejne zapytanie pobiera kolumny `prd_id`, `prd_type_id` oraz `name` z tabeli `more_products`:

```

SELECT prd_id, prd_type_id, name
FROM more_products;

```

PRD_ID	PRD_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Chemia
3		Supernowa
4	2	Lądownie na Księżyco
5	2	Łódź podwodna

Operator UNION ALL

Operator `UNION ALL` zwraca wszystkie wiersze pobrane przez zapytania, łącznie z tymi powtarzającymi się. W poniższym zapytaniu użyto operatora `UNION ALL`:

```

SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products;

```

```
PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
1          1 Nauka współczesna
2          1 Chemia
3          2 Supernowa
4          2 Wojny czołgów
5          2 Z Files
6          2 2412: Powrót
7          3 Space Force 9
8          3 Z innej planety
9          4 Muzyka klasyczna
10         4 Pop 3
11         4 Twórczy wrzask
12         Pierwsza linia
1          1 Nauka współczesna
2          1 Chemia
3          Supernowa
4          2 Lądowanie na Księżyku
5          2 Łódź podwodna
```

17 rows selected.

W zestawie wyników znajdują się wszystkie wiersze z tabel products i more_products razem z tymi powtarzającymi się.

Możemy posortować wiersze, używając klauzuli ORDER BY i pozycji kolumny. W poniższym przykładzie użyto klauzuli ORDER BY 1, aby posortować wiersze według pierwszej kolumny pobranej przez dwa zapytania (product_id i prd_id):

```
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products
ORDER BY 1;
```

```
PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
1          1 Nauka współczesna
1          1 Nauka współczesna
2          1 Chemia
2          1 Chemia
3          2 Supernowa
3          Supernowa
4          2 Wojny czołgów
4          2 Lądowanie na Księżyku
5          2 Łódź podwodna
5          2 Z Files
6          2 2412: Powrót
7          3 Space Force 9
8          3 Z innej planety
9          4 Muzyka klasyczna
10         4 Pop 3
11         4 Twórczy wrzask
12         Pierwsza linia
```

17 rows selected.

Operator UNION

Operator UNION zwraca jedynie niepowtarzające się wiersze zwrócone przez zapytania. W poniższym przykładzie wykorzystano operator UNION:

```
SELECT product_id, product_type_id, name
FROM products
UNION
```

```
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
1          1 Nauka współczesna
2          1 Chemia
3          2 Supernowa
3          Supernowa
4          2 Łądownie na Księżyco
4          2 Wojny czołgów
5          2 Z Files
5          2 Łódź podwodna
6          2 2412: Powrót
7          3 Space Force 9
8          3 Z innej planety
9          4 Muzyka klasyczna
10         4 Pop 3
11         4 Twórczy wrzask
12         Pierwsza linia
```

15 rows selected.

Powtarzające się wiersze „Nauka współczesna” oraz „Chemia” nie zostały pobrane dwukrotnie, więc w wynikach pojawiło się jedynie 15 wierszy.

Operator INTERSECT

Operator INTERSECT zwraca jedynie te wiersze, które zostały pobrane przez obydwa zapytania. W poniższym przykładzie użyto operatora INTERSECT:

```
SELECT product_id, product_type_id, name
FROM products
INTERSECT
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
1          1 Nauka współczesna
2          1 Chemia
```

Zostały zwrócone wiersze „Nauka współczesna” i „Chemia”.

Operator MINUS

Operator MINUS zwraca wiersze pozostałe po odjęciu tych pobranych przez drugie zapytanie od tych pobranych przez pierwsze zapytanie. W poniższym przykładzie zastosowano operator MINUS:

```
SELECT product_id, product_type_id, name
FROM products
MINUS
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
3          2 Supernowa
4          2 Wojny czołgów
5          2 Z Files
6          2 2412: Powrót
7          3 Space Force 9
8          3 Z innej planety
9          4 Muzyka klasyczna
```

```

10          4 Pop 3
11          4 Twórczy wrzask
12          Pierwsza linia

```

10 rows selected.

Wiersze znajdujące się w tabeli `more_products` zostały odjęte od wierszy znajdujących się w tabeli `products` i zostały zwrócone w wynikach zapytania.

Łączenie operatorów zestawu

Za pomocą kilku operatorów zestawu możemy połączyć z sobą więcej niż dwa zapytania — wyniki z jednego operatora będą przekazywane do następnego. Domyslnie operatory zestawu są obliczane z góry do dołu, ale należy określić kolejność ich wykonywania za pomocą nawiasów, na wypadek gdyby firma Oracle Corporation zmieniła domyślne zachowanie w przyszłych wersjach oprogramowania.

W przykładach w tym podrozdziale jest wykorzystywana następująca tabela `product_changes` (tworzona przez skrypt `store_schema.sql`):

```

CREATE TABLE product_changes (
    product_id INTEGER
        CONSTRAINT prod_changes_pk PRIMARY KEY,
    product_type_id INTEGER
        CONSTRAINT prod_changes_fk_product_types
            REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);

```

Poniższe zapytanie zwraca kolumny `product_id`, `product_type_id` oraz `name` z tabeli `product_changes`:

```

SELECT product_id, product_type_id, name
FROM product_changes;

```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Nowa chemia
3	1	Supernowa
13	2	Lądownie na Księżyco
14	2	Łódź podwodna
15	2	Samolot

W kolejnym zapytaniu:

- Operator `UNION` łączy wyniki z tabel `products` i `more_products`. (Zwraca jedynie niepowtarzające się wiersze pobrane przez zapytania).
- Operator `INTERSECT` łączy wyniki z wcześniejszego operatora `UNION` z wynikami z tabeli `product_changes`. (Operator `INTERSECT` zwraca jedynie wyniki pobrane przez obydwa zapytania).
- Nawiąsy są używane do określenia kolejności obliczeń, czyli (1) dla `UNION` między tabelami `products` i `more_products`; (2) dla `INTERSECT`:

```

(SELECT product_id, product_type_id, name
FROM products
UNION
SELECT prd_id, prd_type_id, name
FROM more_products)
INTERSECT
SELECT product_id, product_type_id, name
FROM product_changes;

```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna

W kolejnym zapytaniu ustwiono nawiasy tak, że działanie INTERSECT jest wykonywane jako pierwsze. Zostały zwrócone inne wyniki niż we wcześniejszym przykładzie:

```
SELECT product_id, product_type_id, name
FROM products
UNION
(SELECT prd_id, prd_type_id, name
FROM more_products
INTERSECT
SELECT product_id, product_type_id, name
FROM product_changes);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Chemia
3	2	Supernowa
4	2	Wojny czołgów
5	2	Z Files
6	2	2412: Powrót
7	3	Space Force 9
8	3	Z innej planety
9	4	Muzyka klasyczna
10	4	Pop 3
11	4	Twórczy wrzask
12		Pierwsza linia

To podsumowuje opis operatorów zestawu.

Użycie funkcji TRANSLATE()

Funkcja TRANSLATE(*x, napis_z, napis_do*) konwertuje w *x* wystąpienia znaków z napisu *napis_z* na odpowiednie znaki w napisie *napis_do*. Najlepszym sposobem na zrozumienie działania tej funkcji jest prześledzenie kilku przypadków jej użycia.

W poniższym przykładzie użyto funkcji TRANSLATE() do przesunięcia każdego znaku w napisie TAJNA WIADOMOŚĆ: SPOTKAJMY SIĘ W PARKU o cztery pozycje w prawo (stosujemy polskie znaki diakrytyczne): A staje się Ć, B staje się E itd.:

```
SELECT TRANSLATE('TAJNA WIADOMOŚĆ: SPOTKAJMY SIĘ W PARKU',
'AĄBCĆDĘFGHIJKŁMNOÓPRSŠTUWXYZŻŻ',
'ĆDĘFGHIJKŁMNOÓPRSŠTUWXYZŻŻAĄBC')
FROM dual;
```

```
TRANSLATE('TAJNAWIADOMOŚĆ:SPOTKAJMYSIĘWPARKU',
```

```
YĆMRĆ ŹŁĆGSPSXFX: WTSYNĆMPA WLI Ž TČUNZ
```

Kolejny przykład przesuwa w powyższym wyniku znaki o cztery miejsca w lewo: Ć staje się A, E staje się B itd.:

```
SELECT TRANSLATE('YĆMRĆ ŹŁĆGSPSXFX: WTSYNĆMPA WLI Ž TČUNZ',
'ĆDĘFGHIJKŁMNOÓPRSŠTUWXYZŻŻAĄBC',
'AĄBCĆDĘFGHIJKŁMNOÓPRSŠTUWXYZŻŻ')
FROM dual;
```

```
TRANSLATE('XCLPCZLCFRÓRWE:UŠRXMCŁÓŽULHZŠC
```

```
TAJNA WIADOMOŚĆ: SPOTKAJMY SIĘ W PARKU
```

Do funkcji TRANSLATE() można oczywiście przesyłać wartości z kolumn. W poniższym przykładzie przesłano kolumnę *name* tabeli *products* do funkcji TRANSLATE(), która przesuwa litery w nazwie produktu o cztery miejsca w prawo:

```
SELECT product_id, TRANSLATE(name,
  'AĄBCĆDEĘFGHĲJKLŁMNÓÓPRSŠTUWXYZŽAąbcćdeęfghijkłłmnoóprsštuwxyzż',
  'CДЕĘFGHĲJKLŁMNÓÓPRSŠTUWXYZŽAАБСĆдеęfghijkłłmnoóprsštuwxyzžаabc')
FROM products;
```

```
PRODUCT_ID TRANSLATE(NAME,'AĄBCĆDEĘFGHĲJKLŁMN
```

```
-----  
1 Róznač Žwtśoęąhwrc  
2 Ełhpłć  
3 Wzthurszć  
4 Žsmra ęasóksz  
5 A Jłohw  
6 2412: Tsžušy  
7 Wtćeh Jsuęh 9  
8 A łrrhm toćrhyh  
9 Pzjanč noćwaęarč  
10 Tst 3  
11 Yžśuęąą žuaćwn  
12 Tłhuźwąć ołrlć
```

Z pomocą funkcji `TRANSLATE()` można również konwertować liczby. Poniższy przykład pobiera liczbę 12345 i zmienia 5 na 6, 4 na 7, 3 na 8, 2 na 9 i 1 na 0:

```
SELECT TRANSLATE(12345,
  54321,
  67890)
FROM dual;
```

```
TRANS
-----
09876
```

Użycie funkcji `DECODE()`

Funkcja `DECODE(wartość, szukana_wartość, wynik, domyślna_wartość)` porównuje *wartość* z *szukaną_wartość*. Jeżeli są one równe, funkcja `DECODE()` zwraca *wynik*; w przeciwnym razie zwraca *domyślna_wartość*. Pozwala na wykonywanie logiki `if-then-else` w SQL bez konieczności stosowania PL/SQL. Każdy z jej argumentów może być kolumną, literałem, funkcją lub podzapytaniem.



Funkcja `DECODE()` jest autorskim rozwiązaniem firmy Oracle. Używając Oracle Database 9i oraz nowszych wersji, należy korzystać z wyrażeń `CASE`, które będą omówione w kolejnych podrozdziałach. Z użyciem funkcji `DECODE()` można spotkać się, korzystając ze starszych baz danych Oracle — dlatego została ona tutaj omówiona.

Poniższy przykład obrazuje użycie funkcji `DECODE()` z literałami. Funkcja zwraca 2 (1 jest porównywane z 1, a ponieważ są równe, zwracane jest 2):

```
SELECT DECODE(1, 1, 2, 3)
FROM dual;
```

```
DECODE(1,1,2,3)
-----
```

```
2
```

W kolejnym przykładzie funkcja `DECODE()` porównuje 1 z 2, a ponieważ te liczby są różne, jest zwrotnieana wartość 3:

```
SELECT DECODE(1, 2, 1, 3)
FROM dual;
```

```
DECODE(1,2,1,3)
-----
```

```
3
```

W kolejnym przykładzie jest porównywana kolumna `available` z tabeli `more_products`. Jeżeli `available` jest równe Y, jest zwracany napis 'Produkt jest dostępny'. W przeciwnym razie jest zwracany napis 'Produkt jest niedostępny':

```
SELECT prd_id, available,
       DECODE(available, 'Y', 'Produkt jest dostępny',
              'Produkt jest niedostępny')
  FROM more_products;
```

```
PRD_ID A DECODE(AVAILABLE,'Y','PRO
```

```
-----  
1 Y Produkt jest dostępny  
2 Y Produkt jest dostępny  
3 N Produkt jest niedostępny  
4 N Produkt jest niedostępny  
5 Y Produkt jest dostępny
```

Do funkcji `DECODE()` można przesyłać wiele parametrów wyszukiwania i wyniku, co obrazuje poniższy przykład, w którym kolumna `product_type_id` jest zwracana jako nazwa rodzaju produktu:

```
SELECT product_id, product_type_id,
       DECODE(product_type_id,
              1, 'Książka',
              2, 'VHS',
              3, 'DVD',
              4, 'CD',
              'Czasopismo')
  FROM products;
```

```
PRODUCT_ID PRODUCT_TYPE_ID DECODE(PRO
```

```
-----  
1           1 Książka  
2           1 Książka  
3           2 VHS  
4           2 VHS  
5           2 VHS  
6           2 VHS  
7           3 DVD  
8           3 DVD  
9           4 CD  
10          4 CD  
11          4 CD  
12          Czasopismo
```

Należy zauważyć, że:

- jeżeli `product_type_id` ma wartość 1, zwracane jest wyrażenie `Książka`,
- jeżeli `product_type_id` ma wartość 2, zwracane jest wyrażenie `VHS`,
- jeżeli `product_type_id` ma wartość 3, zwracane jest wyrażenie `DVD`,
- jeżeli `product_type_id` ma wartość 4, zwracane jest wyrażenie `CD`,
- jeżeli `product_type_id` ma jakąkolwiek inną wartość, zwracane jest wyrażenie `Czasopismo`.

Użycie wyrażenia CASE

Wyrażenie `CASE` wykonuje logikę `if-then-else` w instrukcji SQL i jest obsługiwane w Oracle Database 9i i nowszych. Działa podobnie jak funkcja `DECODE()`, ale należy go używać, ponieważ jest ono zgodne ze standardem ANSI SQL/92. Ponadto wyrażenia `CASE` są czytelniejsze.

Wyróżniamy dwa rodzaje wyrażeń `CASE`:

- proste wyrażenia `CASE`, w których do określenia zwracanej wartości są wykorzystywane wyrażenia,
- przeszukiwane wyrażenia `CASE`, w których do określenia zwracanej wartości są wykorzystywane warunki.

Poniżej zostaną opisane obydwa rodzaje wyrażeń CASE.

Proste wyrażenia CASE

W prostych wyrażeniach CASE do określenia zwracanej wartości są wykorzystywane osadzone wyrażenia. Proste wyrażenia CASE mają następującą składnię:

```
CASE przeszukiwane_wyrażenie
WHEN wyrażenie1 THEN wynik1
WHEN wyrażenie2 THEN wynik2
...
WHEN wyrażenieN THEN wynikN
ELSE domyślny_wynik
END
```

Powyżej:

- *przeszukiwane_wyrażenie* jest szacowanym wyrażeniem,
- *wyrażenie1, wyrażenie2, ..., wyrażenieN* są wyrażeniami, które powinny być porównywane z *przeszukiwane_wyrażenie*,
- *wynik1, wynik2, ..., wynikN* są zwracanymi wynikami (po jednym dla każdego możliwego wyrażenia): jeżeli *wyrażenie1* ma wartość taką jak *przeszukiwane_wyrażenie*, jest zwracany *wynik1* i podobnie dla pozostałych wyrażeń,
- *domyślny_wynik* jest zwracany, gdy nie zostanie znalezione pasujące wyrażenie.

W poniższym przykładzie przedstawiono proste wyrażenie CASE, zwracające rodzaje produktów jako nazwy:

```
SELECT product_id, product_type_id,
CASE product_type_id
WHEN 1 THEN 'Książka'
WHEN 2 THEN 'VHS'
WHEN 3 THEN 'DVD'
WHEN 4 THEN 'CD'
ELSE 'Czasopismo'
END
FROM products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	CASEPRODU
1		1 Książka
2		1 Książka
3		2 VHS
4		2 VHS
5		2 VHS
6		2 VHS
7		3 DVD
8		3 DVD
9		4 CD
10		4 CD
11		4 CD
12		Czasopismo

Przeszukiwane wyrażenia CASE

W przeszukiwanych wyrażeniach CASE do określenia zwracanej wartości są wykorzystywane warunki. Przeszukiwane wyrażenia CASE mają następującą składnię:

```
CASE
WHEN warunek1 THEN wynik1
WHEN warunek2 THEN wynik2
...
...
```

180 Oracle Database 12c i SQL. Programowanie

```
WHEN warunekN THEN wynikN
ELSE domyślny_wynik
END
```

Powyżej:

- *warunek1, warunek2, ..., warunekN* są szacowanymi wyrażeniami,
- *wynik1, wynik2, ..., wynikN* są zwracanymi wynikami (po jednym dla każdego warunku): jeżeli *warunek1* jest spełniony, jest zwracany *wynik1* i podobnie dla pozostałych warunków,
- *domyślny_wynik* jest zwracany, jeżeli żaden z warunków nie jest spełniony.

Poniższy przykład ilustruje użycie przeszukiwanego wyrażenia CASE:

```
SELECT product_id, product_type_id,
CASE
    WHEN product_type_id = 1 THEN 'Książka'
    WHEN product_type_id = 2 THEN 'VHS'
    WHEN product_type_id = 3 THEN 'DVD'
    WHEN product_type_id = 4 THEN 'CD'
    ELSE 'Czasopismo'
END
FROM products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	CASEWHENPR
1	1	Książka
2	1	Książka
3	2	VHS
4	2	VHS
5	2	VHS
6	2	VHS
7	3	DVD
8	3	DVD
9	4	CD
10	4	CD
11	4	CD
12		Czasopismo

W przeszukiwanym wyrażeniu CASE można również używać operatorów, co obrazuje poniższy przykład:

```
SELECT product_id, price,
CASE
    WHEN price > 15 THEN 'Drogi'
    ELSE 'Tani'
END
FROM products;
```

PRODUCT_ID	PRICE	CASEW
1	19,95	Drogi
2	30	Drogi
3	25,99	Drogi
4	13,95	Tani
5	49,99	Drogi
6	14,95	Tani
7	13,49	Tani
8	12,99	Tani
9	10,99	Tani
10	15,99	Drogi
11	14,99	Tani
12	13,49	Tani

Bardziej zaawansowane przykłady wyrażeń CASE zostaną przedstawione w dalszej części tego rozdziału.

Zapytania hierarchiczne

Bardzo często możemy spotkać się z danymi uporządkowanymi hierarchicznie. Przykładami mogą być:

- struktura organizacyjna przedsiębiorstwa,
- drzewo rodzinne.

W tym podrozdziale poznasz zapytania, które uzyskują dostęp do hierarchii pracowników naszego fikcyjnego sklepu.

Przykładowe dane

W tym podrozdziale będziemy wykorzystywali tabelę `more_employees`, tworzoną przez skrypt `store_schema.sql` w następujący sposób:

```
CREATE TABLE more_employees (
    employee_id INTEGER
        CONSTRAINT more_employees_pk PRIMARY KEY,
    manager_id INTEGER
        CONSTRAINT more_employees_fk_fk_more_empl
            REFERENCES more_employees(employee_id),
    first_name VARCHAR2(10) NOT NULL,
    last_name VARCHAR2(10) NOT NULL,
    title VARCHAR2(20),
    salary NUMBER(6, 0)
);
```

Kolumna `manager_id` jest odwołaniem wewnętrzny do kolumny `employee_id` tabeli `more_employees`; `manager_id` określa przełożonego danego pracownika (jeżeli taki jest).

Poniższe zapytanie zwraca wiersze z tabeli `more_employees`:

```
SELECT *
FROM more_employees;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1		Jan	Kowalski	Prezes zarządu	800000
2	1	Roman	Joźwierz	Kierownik sprzedaży	600000
3	2	Fryderyk	Helc	Sprzedawca	200000
4	1	Zuzanna	Nowak	Kierownik wsparcia	500000
5	2	Robert	Zielony	Sprzedawca	40000
6	4	Joanna	Brąz	Pracownik wsparcia	45000
7	4	Jan	Szary	Kierownik wsparcia	30000
8	7	Jadwiga	Niebieska	Pracownik wsparcia	29000
9	6	Henryk	Borny	Pracownik wsparcia	30000
10	1	Kazimierz	Czarny	Kierownik obsługi	100000
11	10	Kamil	Długi	Pracownik obsługi	50000
12	10	Franciszek	Słaby	Pracownik obsługi	45000
13	10	Dorota	Prewał	Pracownik obsługi	47000

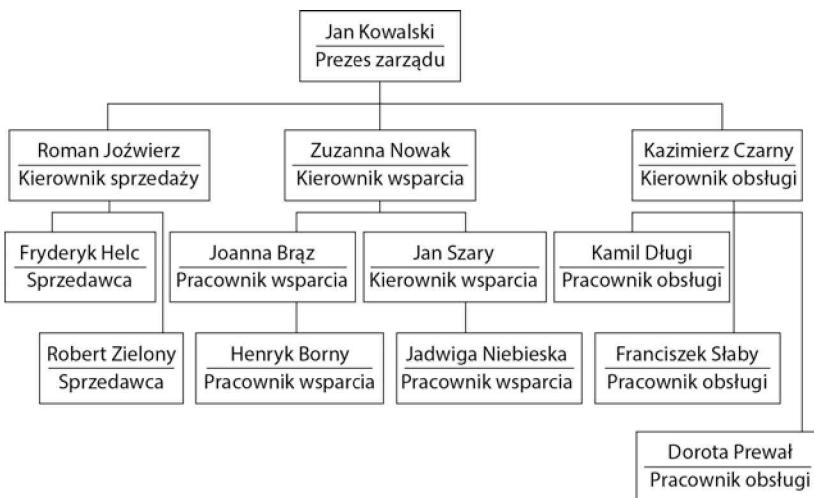
Uchwycenie relacji między pracownikami jest trudne. Zostały one przedstawione w formie graficznej na rysunku 7.1.

Jak widać na rysunku 7.1, elementy — czyli **węzły** — tworzą drzewo. Z drzewami węzłów wiążą się poniższe terminy techniczne:

- **Węzeł główny** znajduje się na samej górze drzewa. W przykładzie przedstawionym na rysunku 7.1 węzłem głównym jest Jan Kowalski, prezes zarządu.
- **Węzeł nadzędny (macierzysty)** to ten, pod którym znajduje się przynajmniej jeden węzeł. Na przykład Jan Kowalski jest węzłem macierzystym dla węzłów Roman Joźwierz, Zuzanna Nowak i Kazimierz Czarny.

Rysunek 7.1.

Relacje między pracownikami



- **Węzeł podrzędny (potomek)** to ten, nad którym znajduje się jeden węzeł macierzysty. Na przykład węzłem macierzystym Romana Joźwierza jest Jan Kowalski.
- **Liść** jest węzłem, który nie posiada potomków. Na przykład Fryderyk Helc i Robert Zielony są węzłami liśćmi.
- **Węzły równorzędne (rodzeństwo)** to węzły mające ten sam węzeł macierzysty. Na przykład Roman Joźwierz, Zuzanna Nowak i Kazimierz Czarny są węzłami równorzędnymi, a ich węzłem macierzystym jest Jan Kowalski. Tak samo Joanna Brąz i Jan Szary są rodzeństwem, a ich węzłem macierzystym jest Zuzanna Nowak.

Jak zostanie za chwilę pokazane, do wykonywania zapytań hierarchicznych służą klauzule CONNECT BY i START WITH instrukcji SELECT. Następnie dowiesz się, jak wykonywać zapytania hierarchiczne za pomocą klauzuli WITH i rekurencyjnych podzapytań przygotowywanych (ta funkcjonalność została wprowadzona w Oracle Database 11g Release 2).

Zastosowanie klauzul CONNECT BY i START WITH

Składnia klauzul CONNECT BY i START WITH instrukcji SELECT jest następująca:

```

SELECT [LEVEL], kolumna, wyrażenie, ...
FROM tabela
[WHERE klauzula_where]
[[START WITH warunek_rozpoczęcia] [CONNECT BY PRIOR warunek_poprzednika]];
  
```

Powyżej:

- LEVEL jest pseudokolumną określającą, na którym poziomie drzewa się znajdujemy. Zwraca 1 dla węzła głównego, 2 dla potomka węzła głównego itd.
- warunek_rozpoczęcia określa, skąd rozpocząć zapytanie hierarchiczne. Przykładem warunek_rozpoczęcia jest employee_id = 1, co określa, że zapytanie rozpoczyna się od pracownika nr 1.
- warunek_poprzednika określa relację między węzłami macierzystymi i potomnymi. Przykładem warunek_rozpoczęcia jest employee_id = manager_id, co określa, że relacja zachodzi między macierzystym employee_id i potomnym manager_id — czyli manager_id potomka wskazuje na employee_id rodzica.

Poniższe zapytanie obrazuje wykorzystanie klauzul START WITH i CONNECT BY PRIOR. Należy zauważać, że pierwszy wiersz zawiera informacje o Janie Kowalskim (pracowniku nr 1), drugi o Romanie Joźwierzu, dla którego manager_id wynosi 1, itd.:

```
SELECT employee_id, manager_id, first_name, last_name
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME
1		Jan	Kowalski
2	1	Roman	Joźwierz
3	2	Fryderyk	Helc
5	2	Robert	Zielony
4	1	Zuzanna	Nowak
6	4	Joanna	Brąz
9	6	Henryk	Borny
7	4	Jan	Szary
8	7	Jadwiga	Niebieska
10	1	Kazimierz	Czarny
11	10	Kamil	Długi
12	10	Franciszek	Słaby
13	10	Dorota	Prewał

Użycie pseudokolumny LEVEL

Kolejne zapytanie obrazuje wykorzystanie pseudokolumny LEVEL do wyświetlenia poziomu w drzewie:

```
SELECT LEVEL, employee_id, manager_id, first_name, last_name
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id
ORDER BY LEVEL;
```

LEVEL	EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME
1	1		Jan	Kowalski
2	10	1	Kazimierz	Czarny
2	2	1	Roman	Joźwierz
2	4	1	Zuzanna	Nowak
3	13	10	Dorota	Prewał
3	7	4	Jan	Szary
3	11	10	Kamil	Długi
3	5	2	Robert	Zielony
3	3	2	Fryderyk	Helc
3	12	10	Franciszek	Słaby
3	6	4	Joanna	Brąz
4	8	7	Jadwiga	Niebieska
4	9	6	Henryk	Borny

W kolejnym zapytaniu użyto funkcji COUNT() do zliczenia poziomów w drzewie:

```
SELECT COUNT(DISTINCT LEVEL)
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;
```

COUNT(DISTINCT LEVEL)
4

4

Formatowanie wyników zapytania hierarchicznego

Wyniki zapytania hierarchicznego możemy sformatować, korzystając z pseudokolumny LEVEL oraz funkcji LPAD(), która dopełnia znakami lewą stroną wartości.

W poniższym wyrażeniu użyto LPAD(' ', 2 * LEVEL - 1) do dopełnienia lewej strony 2 * LEVEL - 1 spacjami. W wyniku tego imię i nazwisko pracownika zostanie wcięte na podstawie wartości LEVEL (czyli LEVEL 1 nie będzie wcięty, LEVEL 2 będzie wcięty o dwie spacje, LEVEL 3 — o cztery spacje itd.):

```
SET PAGESIZE 999
COLUMN employee FORMAT A25
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 START WITH employee_id = 1
 CONNECT BY PRIOR employee_id = manager_id;

LEVEL EMPLOYEE
-----
1 Jan Kowalski
2 Roman Joźwierz
3 Fryderyk Helc
3 Robert Zielony
2 Zuzanna Nowak
3 Joanna Brąz
4 Henryk Borny
3 Jan Szary
4 Jadwiga Niebieska
2 Kazimierz Czarny
3 Kamil Długi
3 Franciszek Słaby
3 Dorota Prewał
```

W tych wynikach możemy łatwo dostrzec relacje między pracownikami.

Rozpoczynanie od węzła innego niż główny

Odczyt drzewa nie musi być rozpoczynany od głównego węzła — korzystając z klauzuli START WITH, możemy rozpocząć od dowolnego węzła. Poniższe zapytanie rozpoczyna od Zuzanny Nowak. Należy zwrócić uwagę, że LEVEL zwraca 1 dla Zuzanny Nowak, 2 dla Joanny Brąz itd.:

```
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 START WITH last_name = 'Nowak'
 CONNECT BY PRIOR employee_id = manager_id;

LEVEL EMPLOYEE
-----
1 Zuzanna Nowak
2 Joanna Brąz
3 Henryk Borny
2 Jan Szary
3 Jadwiga Niebieska
```

Gdyby w sklepie pracowało kilka osób o tym samym nazwisku, moglibyśmy po prostu użyć employee_id w klauzuli START WITH. Na przykład w poniższym zapytaniu użyto employee_id Zuzanny Nowak, czyli 4:

```
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 START WITH employee_id = 4
 CONNECT BY PRIOR employee_id = manager_id;
```

To zapytanie zwraca takie same wyniki jak poprzednie.

Użycie podzapytania w klauzuli START WITH

W klauzuli START WITH możemy użyć podzapytania. Na przykład w poniższym zapytaniu użyto podzapytania do wybrania employee_id odpowiadającego pracownikowi Kazimierzowi Czarnemu. Wartość tego employee_id jest przesyłana do klauzuli START WITH:

```
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 START WITH employee_id = (
    SELECT employee_id
      FROM more_employees
     WHERE first_name = 'Kazimierz'
       AND last_name = 'Czarny'
)
 CONNECT BY PRIOR employee_id = manager_id;

LEVEL EMPLOYEE
-----
1  Kazimierz Czarny
2   Kamil Długi
2   Franciszek Słaby
2   Dorota Prewał
```

Poruszanie się po drzewie w góre

Nie musimy poruszać się po drzewie w dół, od rodziców do potomków. Możemy rozpocząć od potomka i przesuwać się w górę. W tym celu należy zamienić miejscami potomka i rodzica w klauzuli CONNECT BY PRIOR. Na przykład CONNECT BY PRIOR manager_id = employee_id łączy manager_id potomka z employee_id rodzica.

Poniższe zapytanie rozpoczyna pracę od Jadwigi Niebieskiej i porusza się w górę, aż do Jana Kowalskiego. LEVEL zwraca 1 dla Jadwigi Niebieskiej, 2 dla Jana Szarego itd.:

```
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 START WITH last_name = 'Niebieska'
 CONNECT BY PRIOR manager_id = employee_id;

LEVEL EMPLOYEE
-----
1  Jadwiga Niebieska
2   Jan Szary
3   Zuzanna Nowak
4   Jan Kowalski
```

Eliminowanie węzłów i gałęzi z zapytania hierarchicznego

Z pomocą klauzuli WHERE możemy usunąć określony wiersz z drzewa zapytania. W poniższym zapytaniu usunięto z wyników dane o Romanie Joźwierzu, stosując klauzulę WHERE last_name != 'Joźwierz':

```
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
  FROM more_employees
 WHERE last_name != 'Joźwierz'
 START WITH employee_id = 1
 CONNECT BY PRIOR employee_id = manager_id;

LEVEL EMPLOYEE
-----
```

```

1 Jan Kowalski
3 Fryderyk Helc
3 Robert Zielony
2 Zuzanna Nowak
3 Joanna Brąz
4 Henryk Borny
3 Jan Szary
4 Jadwiga Niebieska
2 Kazimierz Czarny
3 Kamil Długi
3 Franciszek Słaby
3 Dorota Prewał

```

Zauważmy, że choć dane Romana Joźwierza zostały usunięte z wyników, jego pracownicy — Fryderyk Helc i Robert Zielony — wciąż znajdują się na liście.

Aby usunąć całą gałąź węzłów z wyników, należy dodać klauzulę **AND** do klauzuli **CONNECT BY PRIOR**. Na przykład w poniższym wyrażeniu użyto klauzuli **AND last_name != 'Joźwierz'**, aby usunąć z wyników dane Romana Joźwierza i wszystkich jego podwładnych:

```

SELECT LEVEL,
LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'Joźwierz';

```

LEVEL EMPLOYEE

```

-----
1 Jan Kowalski
2 Zuzanna Nowak
3 Joanna Brąz
4 Henryk Borny
3 Jan Szary
4 Jadwiga Niebieska
2 Kazimierz Czarny
3 Kamil Długi
3 Franciszek Słaby
3 Dorota Prewał

```

Umieszczanie innych warunków w zapytaniu hierarchicznym

Korzystając z klauzuli **WHERE**, możemy umieścić inne warunki w zapytaniu hierarchicznym. W poniższym przykładzie użyto klauzuli **WHERE** do pobrania informacji o pracownikach, których pensje wynoszą 50 000 zł lub są niższe:

```

SELECT LEVEL,
LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee,
salary
FROM more_employees
WHERE salary <= 50000
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;

```

LEVEL EMPLOYEE

SALARY

```

-----
3 Robert Zielony      40000
3 Joanna Brąz         45000
4 Henryk Borny       30000
3 Jan Szary          30000
4 Jadwiga Niebieska 29000
3 Kamil Długi        50000
3 Franciszek Słaby   45000
3 Dorota Prewał     47000

```

Zapytania hierarchiczne wykorzystujące rekurencyjne podzapytania przygotowywane

Wsparcie rekurencyjnych podzapytań przygotowywanych (ang. *recursive subquery factoring*) pojawiło się w Oracle Database 11g Release 2. Funkcjonalność ta umożliwia tworzenie zapytań o dane hierarchiczne i jest potężniejsza niż CONNECT BY, ponieważ umożliwia przeszukiwanie w głąb i przeszukiwanie wszerz oraz wspiera wielokrotnie zagębione gałęzie.

Podzapytanie umieszcza się w klauzuli WITH i można się do niego odwoływać spoza tej klauzuli. Poniższy przykład pokazuje zastosowanie rekurencyjnego podzapytania przygotowanego do wyświetlenia hierarchii pracowników i poziomów raportowania. Ten przykład zawiera podzapytanie nazwane `reporting_hierarchy` umieszczone wewnątrz klauzuli WITH. Głównie zapytanie spoza klauzuli WITH przetwarza wyniki z podzapytania `reporting_hierarchy`.

```
WITH
  reporting_hierarchy (
    employee_id, manager_id, reporting_level, first_name, last_name
  ) AS (
    SELECT
      employee_id, manager_id, 0 reporting_level, first_name, last_name
    FROM more_employees
    WHERE employee_id = 1
    UNION ALL
    SELECT
      e.employee_id, e.manager_id, reporting_level + 1,
      e.first_name, e.last_name
    FROM reporting_hierarchy r, more_employees e
    WHERE r.employee_id = e.manager_id
  )
SELECT
  employee_id, manager_id, reporting_level, first_name, last_name
FROM reporting_hierarchy
ORDER BY employee_id;
```

EMPLOYEE_ID	MANAGER_ID	REPORTING_LEVEL	FIRST_NAME	LAST_NAME
1		0	Jan	Kowalski
2	1	1	Roman	Joźwierz
3	2	2	Fryderyk	Helc
4	1	1	Zuzanna	Nowak
5	2	2	Robert	Zielony
6	4	2	Joanna	Brąz
7	4	2	Jan	Szary
8	7	3	Jadwiga	Niebieska
9	6	3	Henryk	Borny
10	1	1	Kazimierz	Czarny
11	10	2	Kamil	Długi
12	10	2	Franciszek	Słaby
13	10	2	Dorota	Prewał

Pierwsza instrukcja SELECT zawiera `reporting_level` równe 0, co oznacza, że najwyższy poziom raportowania jest oznaczony tą wartością. Oznacza to też, że w wynikach poziom raportowania dla Jana Kowalskiego (prezesa zarządu) jest ustawiony na zero. Kolejne zapytanie SELECT ma `reporting_level + 1`, co zwiększa o jeden poziom raportowania w hierarchii. Oznacza to, że poziom raportowania dla Romana Joźwierza, Zuzanny Nowak i Kazimierza Czarnego w wynikach jest ustawiony na 1 (ci pracownicy raportują bezpośrednio do Jana Kowalskiego). Poziom raportowania dla innych pracowników jest ustawiony odpowiednio do ich pozycji w hierarchii.

Wiersze równorzędne mają ten sam wiersz macierzysty. Wyszukiwanie w głąb zwraca węzły podzielone przed równorzędnymi. Wyszukiwanie wszerz zwraca wiersze równorzędne przed wierszami potomnymi. Poniższy przykład pokazuje wyszukiwanie w głąb pracowników i formuluje wyniki w taki sposób, by hierarchia raportowania była przejrzysta:

```

WITH
  reporting_hierarchy (
    employee_id, manager_id, reporting_level, first_name, last_name
  ) AS (
    SELECT
      employee_id, manager_id, 0 reporting_level, first_name, last_name
    FROM more_employees
    WHERE manager_id IS NULL
  UNION ALL
    SELECT
      e.employee_id, e.manager_id,
      r.reporting_level + 1 AS reporting_level,
      e.first_name, e.last_name
    FROM reporting_hierarchy r, more_employees e
    WHERE r.employee_id = e.manager_id
  )
SEARCH DEPTH FIRST BY employee_id SET order_by_employee_id
SELECT
  employee_id, manager_id, reporting_level,
  lpad(' ', 2 * reporting_level) || first_name ||
  ' ' || last_name AS name
FROM reporting_hierarchy
ORDER BY order_by_employee_id;

```

EMPLOYEE_ID MANAGER_ID REPORTING_LEVEL NAME

1		0	Jan Kowalski
2	1	1	Roman Joźwierz
3	2	2	Fryderyk Helc
5	2	2	Robert Zielony
4	1	1	Zuzanna Nowak
6	4	2	Joanna Brąz
9	6	3	Henryk Borny
7	4	2	Jan Szary
8	7	3	Jadwiga Niebieska
10	1	1	Kazimierz Czarny
11	10	2	Kamil Długi
12	10	2	Franciszek Słaby
13	10	2	Dorota Prewał

Dla Romana Jóźwierza, Zuzanny Nowak i Kazimierza Czarnego, którzy są węzłami równorzędnymi, węzłem nadzującym jest Jan Kowalski. Tak samo Joanna Brąz i Jan Szary są węzłami równorzędnymi, których węzłem nadzującym jest Zuzanna Nowak.

Wyszukiwanie wszerz zwraca węzły równorzędne przed węzłami podzającymi. Poniższy przykład pokazuje wyszukiwanie wszerz wśród pracowników:

```

WITH
  reporting_hierarchy (
    employee_id, manager_id, reporting_level, first_name, last_name
  ) AS (
    SELECT
      employee_id, manager_id, 0 reporting_level, first_name, last_name
    FROM more_employees
    WHERE manager_id IS NULL
  UNION ALL
    SELECT
      e.employee_id, e.manager_id,
      r.reporting_level + 1 AS reporting_level,
      e.first_name, e.last_name
    FROM reporting_hierarchy r, more_employees e
    WHERE r.employee_id = e.manager_id
  )
SEARCH BREADTH FIRST BY employee_id SET order_by_employee_id

```

```

SELECT
    employee_id, manager_id, reporting_level,
    lpad(' ', 2 * reporting_level) || first_name ||
    ' ' || last_name AS name
FROM reporting_hierarchy
ORDER BY order_by_employee_id;

```

EMPLOYEE_ID MANAGER_ID REPORTING_LEVEL NAME

1		0	Jan Kowalski
2	1	1	Roman Joźwierz
4	1	1	Zuzanna Nowak
10	1	1	Kazimierz Czarny
3	2	2	Fryderyk Helc
5	2	2	Robert Zielony
6	4	2	Joanna Brąz
7	4	2	Jan Szary
11	10	2	Kamil Długi
12	10	2	Franciszek Słaby
13	10	2	Dorota Prewał
8	7	3	Jadwiga Niebieska
9	6	3	Henryk Borny

Poniższy przykład pobiera menedżerów i liczbę pracowników, którymi zarządzają:

```

WITH
reporting_hierarchy (
    employee_id, manager_id, first_name, last_name,
    reporting_level, employee_count
) AS (
    SELECT
        employee_id, manager_id, first_name, last_name,
        0 reporting_level, 0 employee_count
    FROM more_employees
    UNION ALL
    SELECT
        e.employee_id, e.manager_id, e.first_name, e.last_name,
        r.reporting_level + 1 reporting_level, 1 employee_count
    FROM reporting_hierarchy r, more_employees e
    WHERE e.employee_id = r.manager_id
)
SEARCH DEPTH FIRST BY employee_id SET order_by_employee_id
SELECT
    employee_id, manager_id, first_name, last_name,
    SUM(employee_count) AS emp_count,
    MAX(reporting_level) AS rept_level
FROM reporting_hierarchy
GROUP BY employee_id, manager_id, first_name, last_name
HAVING MAX(reporting_level) > 0
ORDER BY employee_id;

```

EMPLOYEE_ID MANAGER_ID FIRST_NAME LAST_NAME EMP_COUNT REPT_LEVEL

1		Jan	Kowalski	12	3
2	1	Roman	Joźwierz	2	1
4	1	Zuzanna	Nowak	4	2
6	4	Joanna	Brąz	1	1
7	4	Jan	Szary	1	1
10	1	Kazimierz	Czarny	3	1

By określić cykle w kolejnych poziomach danych, można wykorzystać klauzulę CYCLE. Poniższy przykład pokazuje wykorzystanie instrukcji CYCLE title SET same_title TO 'T' DEFAULT 'N', która zwraca T w przypadku pracownika mającego takie samo stanowisko jak jego zwierzchnik w hierarchii raportowania:

```

WITH
  reporting_hierarchy (
    employee_id, manager_id, reporting_level,
    first_name, last_name, title
  ) AS (
    SELECT
      employee_id, manager_id, 0 reporting_level,
      first_name, last_name, title
    FROM more_employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT
      e.employee_id, e.manager_id,
      r.reporting_level + 1 AS reporting_level,
      e.first_name, e.last_name, e.title
    FROM reporting_hierarchy r, more_employees e
    WHERE r.employee_id = e.manager_id
  )
SEARCH DEPTH FIRST BY employee_id SET order_by_employee_id
CYCLE title SET same_title TO 'T' DEFAULT 'N'
SELECT
  employee_id AS emp_id, manager_id AS mgr_id,
  lpad(' ', 2 * reporting_level) || first_name ||
  ' ' || last_name AS name,
  title, same_title
FROM reporting_hierarchy
ORDER BY order_by_employee_id;

```

EMP_ID	MGR_ID	NAME	TITLE	S
1		Jan Kowalski	Prezes zarządu	N
2	1	Roman Joźwierz	Kierownik sprzedaży	N
3	2	Fryderyk Helc	Sprzedażca	N
5	2	Robert Zielony	Sprzedażca	N
4	1	Zuzanna Nowak	Kierownik wsparcia	N
6	4	Joanna Brąz	Pracownik wsparcia	N
9	6	Henryk Borny	Pracownik wsparcia	T
7	4	Jan Szary	Kierownik wsparcia	T
10	1	Kazimierz Czarny	Kierownik obsługi	N
11	10	Kamil Długi	Pracownik obsługi	N
12	10	Franciszek Słaby	Pracownik obsługi	N
13	10	Dorota Prewał	Pracownik obsługi	N

To kończy omówienie zapytań hierarchicznych. W kolejnym podrozdziale omówione zostaną klauzule ROLLUP i CUBE.

Klauzule ROLLUP i CUBE

W tym podrozdziale opisano:

- klauzulę ROLLUP, która zwraca wiersz zawierający podsumowanie częściowe dla każdej grupy wierszy oraz wiersz zawierający podsumowanie całkowite dla wszystkich grup;
- klauzulę CUBE, która zwraca wiersze zawierające podsumowania częściowe dla wszystkich połączeń kolumn oraz wiersz zawierający podsumowanie całkowite.

Na początek przyjrzyjmy się przykładowym tabelom wykorzystywanym w tym podrozdziale.

Przykładowe tabele

W podrozdziale będą wykorzystywane następujące tabele, które precyzyjnie reprezentują pracowników sklepu:

- **divisions**, w której są składowane dane o działach przedsiębiorstwa,
- **jobs**, w której są składowane zadania,
- **employees2**, w której są składowane informacje o pracownikach.

Te tabele są tworzone przez skrypt *store_schema.sql*. Tabela **divisions** została utworzona za pomocą poniższej instrukcji:

```
CREATE TABLE divisions (
    division_id CHAR(4)
        CONSTRAINT divisions_pk PRIMARY KEY,
    name VARCHAR2(15) NOT NULL
);
```

Poniższe zapytanie pobiera wszystkie wiersze z tabeli **divisions**:

```
SELECT *
FROM divisions;
```

DIV	NAME
SPR	Sprzedaż
OPE	Obsługa
WSP	Wsparcie
BIZ	Biznes

Tabelę **jobs** tworzy się za pomocą następującej instrukcji:

```
CREATE TABLE jobs (
    job_id CHAR(4)
        CONSTRAINT jobs_pk PRIMARY KEY,
    name VARCHAR2(20) NOT NULL
);
```

Poniższe zapytanie pobiera wiersze z tabeli **jobs**:

```
SELECT *
FROM jobs;
```

JOB	NAME
PRA	Pracownik
KIE	Kierownik
INŻ	Inżynier
TEC	Technolog
PRE	Prezes

Tabela **employees2** została utworzona za pomocą następującej instrukcji:

```
CREATE TABLE employees2 (
    employee_id INTEGER
        CONSTRAINT employees2_pk PRIMARY KEY,
    division_id CHAR(4)
        CONSTRAINT employees2_fk_divisions
            REFERENCES divisions(division_id),
    job_id CHAR(4) REFERENCES jobs(job_id),
    first_name NVARCHAR2(11) NOT NULL,
    last_name NVARCHAR2(11) NOT NULL,
    salary NUMBER(6, 0)
);
```

Poniższe zapytanie pobiera pierwsze pięć wierszy z tabeli **employees2**:

```
SELECT *
FROM employees2
WHERE ROWNUM <= 5;
```

EMPLOYEE_ID	DIV	JOB	FIRST_NAME	LAST_NAME	SALARY

1 BIZ PRE Jan	Kowalski	800000
2 SPR KIE Roman	Joźwierz	350000
3 SPR PRA Fryderyk	Hełc	140000
4 WSP KIE Zuzanna	Nowak	200000
5 SPR PRA Robert	Zielony	350000

Użycie klauzuli ROLLUP

Klauzula ROLLUP rozszerza GROUP BY o zwracanie wiersza zawierającego podsumowanie częściowe każdej grupy wierszy oraz podsumowanie całkowite wszystkich grup.

Klauzula GROUP BY służy do grupowania wierszy w bloki z tą samą wartością we wskazanej kolumnie. Na przykład w poniższym zapytaniu użyto jej do pogrupowania wierszy z tabeli employees2 według department_id. Wykorzystano także funkcję SUM() do obliczenia sumy wynagrodzeń dla każdego division_id:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY division_id
ORDER BY division_id;
```

```
DIV SUM(SALARY)
-----
BIZ    1610000
OPE    1320000
SPR    4936000
WSP    1015000
```

Przesyłanie jednej kolumny do klauzuli ROLLUP

Poprzednie zapytanie zmieniono tak, aby wykorzystać klauzulę ROLLUP. Należy zauważyc, że na końcu pojawił się dodatkowy wiersz zawierający sumę wynagrodzeń we wszystkich grupach:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

```
DIV SUM(SALARY)
-----
BIZ    1610000
OPE    1320000
SPR    4936000
WSP    1015000
          8881000
```



Jeżeli wiersze powinny być uporządkowane w określonej kolejności, należy użyć klauzuli ORDER BY, na wypadek gdyby Oracle Corporation zmieniło domyślną kolejność wierszy zwracanych przez ROLLUP.

Przesyłanie wielu kolumn do klauzuli ROLLUP

Do klauzuli ROLLUP możemy przesyłać wiele kolumn. W takim przypadku wiersze są grupowane w bloki z tymi samymi wartościami kolumn. W poniższym przykładzie przesłano do klauzuli ROLLUP kolumny division_id i job_id tabeli employees2 — wiersze zostaną pogrupowane według tych kolumn:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;
```

```
DIV JOB SUM(SALARY)
-----
BIZ KIE      530000
```

BIZ PRA	280000
BIZ PRE	800000
BIZ	1610000
OPE INŻ	245000
OPE KIE	805000
OPE PRA	270000
OPE	1320000
SPR KIE	4446000
SPR PRA	490000
SPR	4936000
WSP KIE	465000
WSP PRA	435000
WSP TEC	115000
WSP	1015000
	8881000

W wynikach należy zauważyc, że wynagrodzenia zostały podsumowane według `division_id` i `job_id`, a także, że klauzula `ROLLUP` zwróciła wiersz z sumą wynagrodzeń w każdym `division_id` oraz końcowy wiersz z całkowitym podsumowaniem wynagrodzeń

Zmienianie pozycji kolumn przesyłanych do ROLLUP

W kolejnym przykładzie zamieniono miejscami `division_id` i `job_id`. Na skutek tego `ROLLUP` oblicza sumę wynagrodzeń dla każdego `job_id`:

```
SELECT job_id, division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(job_id, division_id)
ORDER BY job_id, division_id;
```

JOB DIV SUM(SALARY)	
---	-----
INŻ OPE	245000
INŻ	245000
KIE BIZ	530000
KIE OPE	805000
KIE SPR	4446000
KIE WSP	465000
KIE	6246000
PRA BIZ	280000
PRA OPE	270000
PRA SPR	490000
PRA WSP	435000
PRA	1475000
PRE BIZ	800000
PRE	800000
TEC WSP	115000
TEC	115000
	8881000

Używanie innych funkcji agregujących z klauzulą ROLLUP

Z klauzulą `ROLLUP` możemy użyć dowolnej funkcji agregującej (lista głównych funkcji agregujących znajduje się w tabeli 4.10 w rozdziale 4.). W poniższym przykładzie użyto funkcji `AVG()` do obliczenia średnich wynagrodzeń:

```
SELECT division_id, job_id, AVG(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;
```

DIV JOB AVG(SALARY)	
---	-----
BIZ KIE	176666,667

```
BIZ PRA      280000
BIZ PRE      800000
BIZ          322000
OPE INŻ      245000
OPE KIE      201250
OPE PRA      135000
OPE          188571,429
SPR KIE      261529,412
SPR PRA      245000
SPR          259789,474
WSP KIE      232500
WSP PRA      145000
WSP TEC      115000
WSP          169166,667
              240027,027
```

Klauzula CUBE

Klauzula CUBE rozszerza klauzułę GROUP BY o zwracanie wierszy zawierających podsumowania częściowe dla wszystkich kombinacji kolumn oraz wiersza zawierającego podsumowanie całkowite. W poniższym przykładzie przesłano do klauzuli CUBE kolumny division_id i job_id. Wiersze zostaną pogrupowane według tych kolumn:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
ORDER BY division_id, job_id;
```

```
DIV JOB SUM(SALARY)
--- -----
BIZ KIE      530000
BIZ PRA      280000
BIZ PRE      800000
BIZ          1610000
OPE INŻ      245000
OPE KIE      805000
OPE PRA      270000
OPE          1320000
SPR KIE      4446000
SPR PRA      490000
SPR          4936000
WSP KIE      465000
WSP PRA      435000
WSP TEC      115000
WSP          1015000
INŻ          245000
KIE          6246000
PRA          1475000
PRE          800000
TEC          115000
              8881000
```

Wynagrodzenia są sumowane według division_id oraz job_id. Klauzula CUBE zwraca wiersz z sumą wynagrodzeń dla każdego division_id, a także (bliżej końca) sumy wszystkich wynagrodzeń dla każdego job_id. Na samym końcu jest zwracane podsumowanie całkowite wynagrodzeń.

W kolejnym przykładzie zamieniono miejscami division_id i job_id:

```
SELECT job_id, division_id, SUM(salary)
FROM employees2
GROUP BY CUBE(job_id, division_id)
ORDER BY job_id, division_id;
```

```
JOB DIV SUM(SALARY)
--- -----
              
```

INŻ OPE	245000
INŻ	245000
KIE BIZ	530000
KIE OPE	805000
KIE SPR	4446000
KIE WSP	465000
KIE	6246000
PRA BIZ	280000
PRA OPE	270000
PRA SPR	490000
PRA WSP	435000
PRA	1475000
PRE BIZ	800000
PRE	800000
TEC WSP	115000
TEC	115000
BIZ	1610000
OPE	1320000
SPR	4936000
WSP	1015000
	8881000

Funkcja GROUPING()

Funkcja GROUPING() przyjmuje kolumnę i zwraca 0 lub 1. Zwraca ona 1, jeżeli wartość kolumny wynosi NULL, a 0, jeśli wartością kolumny nie jest NULL. Funkcja ta jest używana jedynie w zapytaniach wykorzystujących klauzule ROLLUP lub CUBE. Przydaje się, jeżeli chcemy wyświetlić wartość tam, gdzie w przeciwnym razie zostałaby zwrócona wartość NULL.

Użycie funkcji GROUPING() z jedną kolumną w klauzuli ROLLUP

Jak zaprezentowano w podrozdziale „Przesyłanie jednej kolumny do klauzuli ROLLUP”, ostatni wiersz w przykładowym zestawie wyników zawierał sumę wszystkich wynagrodzeń:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

DIV	SUM(SALARY)

BIZ	1610000
OPE	1320000
SPR	4936000
WSP	1015000
	8881000

W kolumnie division_id w ostatnim wierszu znajduje się wartość NULL.

Za pomocą funkcji GROUPING() możemy określić, czy ta kolumna ma wartość NULL (co obrazuje późniejsze zapytanie). Należy zwrócić uwagę, że funkcja GROUPING() zwraca 0 dla wierszy mających wartość division_id różną od NULL i 1 dla ostatniego wiersza, w którym division_id ma wartość NULL:

```
SELECT GROUPING(division_id), division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

GROUPING(DIVISION_ID)	DIV	SUM(SALARY)

0	BIZ	1610000
0	OPE	1320000
0	SPR	4936000
0	WSP	1015000
1		8881000

Użycie wyrażenia CASE do konwersji wartości zwróconej przez funkcję GROUPING()

Za pomocą wyrażenia CASE możemy konwertować wartość 1 z poprzedniego przykładu na bardziej zrozumiałą. Poniżej użyto wyrażenia CASE do konwersji 1 na napis 'Wszystkie działy':

```
SELECT
  CASE GROUPING(division_id)
    WHEN 1 THEN 'Wszystkie działy'
    ELSE division_id
  END AS div,
  SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

DIV	SUM(SALARY)
BIZ	1610000
OPE	1320000
SPR	4936000
WSP	1015000
Wszystkie działy	8881000

Użycie wyrażenia CASE i funkcji GROUPING do konwersji wartości z wielu kolumn

W kolejnym przykładzie rozwinięto pomysł zastępowania wartości NULL, stosując go z klauzulą ROLLUP, zawierającą kilka kolumn (division_id i job_id). Wartości NULL w kolumnie division_id są zastępowane napisem 'Wszystkie działy', a wartości NULL w kolumnie job_id są zastępowane napisem 'Wszystkie zadania':

```
SELECT
  CASE GROUPING(division_id)
    WHEN 1 THEN 'Wszystkie działy'
    ELSE division_id
  END AS div,
  CASE GROUPING(job_id)
    WHEN 1 THEN 'Wszystkie zadania'
    ELSE job_id
  END AS job,
  SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;
```

DIV	JOB	SUM(SALARY)
BIZ	KIE	530000
BIZ	PRA	280000
BIZ	PRE	800000
BIZ	Wszystkie zadania	1610000
OPE	INZ	245000
OPE	KIE	805000
OPE	PRA	270000
OPE	Wszystkie zadania	1320000
SPR	KIE	4446000
SPR	PRA	490000
SPR	Wszystkie zadania	4936000
WSP	KIE	465000
WSP	PRA	435000
WSP	TEC	115000
WSP	Wszystkie zadania	1015000
Wszystkie działy	Wszystkie zadania	8881000

Użycie funkcji GROUPING() z klauzulą CUBE

Funkcji GROUPING() możemy użyć z klauzulą CUBE, tak jak w poniższym przykładzie:

```
SELECT
    CASE GROUPING(division_id)
        WHEN 1 THEN 'Wszystkie działy'
        ELSE division_id
    END AS div,
    CASE GROUPING(job_id)
        WHEN 1 THEN 'Wszystkie zadania'
        ELSE job_id
    END AS job,
    SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
ORDER BY division_id, job_id;
```

DIV	JOB	SUM(SALARY)
BIZ	KIE	530000
BIZ	PRA	280000
BIZ	PRE	800000
BIZ	Wszystkie zadania	1610000
OPE	INŻ	245000
OPE	KIE	805000
OPE	PRA	270000
OPE	Wszystkie zadania	1320000
SPR	KIE	4446000
SPR	PRA	490000
SPR	Wszystkie zadania	4936000
WSP	KIE	465000
WSP	PRA	435000
WSP	TEC	115000
WSP	Wszystkie zadania	1015000
Wszystkie działy	INŻ	245000
Wszystkie działy	KIE	6246000
Wszystkie działy	PRA	1475000
Wszystkie działy	PRE	800000
Wszystkie działy	TEC	115000
Wszystkie działy	Wszystkie zadania	8881000

Klauzula GROUPING SETS

Klauzula GROUPING SETS pozwala uzyskać same wiersze podsumowań częściowych. W poniższym przykładzie użyto jej do pobrania częściowych podsumowań wynagrodzeń według division_id i job_id:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY GROUPING SETS(division_id, job_id)
ORDER BY division_id, job_id;
```

DIVI	JOB_	SUM(SALARY)
BIZ		1610000
OPE		1320000
SPR		4936000
WSP		1015000
INŻ		245000
KIE		6246000
PRA		1475000
PRE		800000
TEC		115000

Zostały zwrócone jedynie podsumowania częściowe dla kolumn `division_id` i `job_id`. Nie zostało zwrócone podsumowanie całkowite wszystkich wynagrodzeń.



Klauzula `GROUPING SETS` jest zwykle bardziej wydajna od `CUBE`, dlatego jeżeli jest to możliwe, należy stosować właśnie ją.

Z kolejnego podrozdziału dowiesz się, jak za pomocą funkcji `GROUPING_ID()` można uzyskać podsumowania częściowe razem z całkowitym.

Użycie funkcji `GROUPING_ID()`

Funkcja `GROUPING_ID()` umożliwia filtrowanie wierszy za pomocą klauzuli `HAVING`, aby wyłączyć wiersze, które nie zawierają podsumowań częściowych lub całkowitych. Przyjmuje ona jedną kolumnę lub więcej i zwraca dziesiętny ekwiwalent wektora bitowego `GROUPING`, który jest obliczany przez połączenie wyników wywołania funkcji `GROUPING()` kolejno dla każdej kolumny.

Obliczanie wektora bitowego `GROUPING`

W podrozdziale „Funkcja `GROUPING()`” pokazano, że funkcja `GROUPING()` zwraca 1, jeżeli wartość kolumny wynosi `NULL`, lub 0, jeśli wartość jest inna niż `NULL`. Na przykład:

- Jeżeli zarówno `division_id`, jak i `job_id` są różne od `NULL`, funkcja `GROUPING()` zwróci 0 dla obydwu kolumn. Wynik dla `division_id` jest łączony z wynikiem dla `job_id` i uzyskujemy wektor bitowy 00, którego odpowiednikiem dziesiętnym jest 0. Dlatego jeżeli zarówno `division_id`, jak i `job_id` są różne od `NULL`, funkcja `GROUPING_ID()` zwróci 0.
- Jeżeli `division_id` jest różne od `NULL` (bit `GROUPING` to 0), ale `job_id` ma wartość `NULL` (bit `GROUPING` to 1), powstaje wektor 01 i funkcja `GROUPING_ID()` zwraca 1.
- Jeżeli `division_id` ma wartość `NULL` (bit `GROUPING` to 1), ale `job_id` ma wartość różną od `NULL` (bit `GROUPING` to 0), powstaje wektor bitowy 10 i funkcja `GROUPING_ID()` zwraca 2.
- Jeżeli zarówno `division_id`, jak i `job_id` mają wartość `NULL` (obydwa bity `GROUPING` to 1), powstaje wektor bitowy 11 i funkcja `GROUPING_ID()` zwraca 3.

Tabela 7.2 zawiera podsumowanie tych wyników.

Tabela 7.2. Wartości zwracane przez funkcję `GROUPING_ID()`

<code>division_id</code>	<code>job_id</code>	Wektor bitowy	Wartość zwracana przez <code>GROUPING_ID()</code>
różna od <code>NULL</code>	różna od <code>NULL</code>	00	0
różna od <code>NULL</code>	<code>NULL</code>	01	1
<code>NULL</code>	różna od <code>NULL</code>	10	2
<code>NULL</code>	<code>NULL</code>	11	3

Przykładowe zapytanie obrazujące użycie funkcji `GROUPING_ID()`

W poniższym przykładzie przesyłano `division_id` i `job_id` do funkcji `GROUPING_ID()`. Należy zauważać, że wyniki zwracane przez tę funkcję są zgodne z przewidywaniami zamieszczonymi w tabeli 7.2:

```

SELECT
  division_id, job_id,
  GROUPING(division_id) AS DIV_GRP,
  GROUPING(job_id) AS JOB_GRP,
  GROUPING_ID(division_id, job_id) AS grp_id,
  SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
ORDER BY division_id, job_id;

```

DIVI JOB_ DIV_GRP JOB_GRP GRP_ID SUM(SALARY)

BIZ	KIE	0	0	0	530000
BIZ	PRA	0	0	0	280000
BIZ	PRE	0	0	0	800000
BIZ		0	1	1	1610000
OPE	INŻ	0	0	0	245000
OPE	KIE	0	0	0	805000
OPE	PRA	0	0	0	270000
OPE		0	1	1	1320000
SPR	KIE	0	0	0	4446000
SPR	PRA	0	0	0	490000
SPR		0	1	1	4936000
WSP	KIE	0	0	0	465000
WSP	PRA	0	0	0	435000
WSP	TEC	0	0	0	115000
WSP		0	1	1	1015000
INŻ		1	0	2	245000
KIE		1	0	2	6246000
PRA		1	0	2	1475000
PRE		1	0	2	800000
TEC		1	0	2	115000
		1	1	3	8881000

Filtrowanie wierszy za pomocą GROUPING_ID() i klauzuli HAVING

Jedno z przydatnych zastosowań funkcji GROUPING_ID() polega na filtrowaniu wierszy za pomocą klauzuli HAVING. Może ona wykluczyć wiersze, które nie zawierają podsumowania częściowego lub całkowitego, przez proste sprawdzenie, czy funkcja GROUPING_ID() zwraca wartość większą od 0. Na przykład:

```
SELECT
    division_id, job_id,
    GROUPING_ID(division_id, job_id) AS grp_id,
    SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
HAVING GROUPING_ID(division_id, job_id) > 0
ORDER BY division_id, job_id;
```

DIVI	JOB_	GRP_ID	SUM(SALARY)
BIZ		1	1610000
OPE		1	1320000
SPR		1	4936000
WSP		1	1015000
INŻ		2	245000
KIE		2	6246000
PRA		2	1475000
PRE		2	800000
TEC		2	115000
		3	8881000

Kilkukrotne użycie kolumny w klauzuli GROUP BY

W klauzuli GROUP BY możemy kilkakrotnie wykorzystać tę samą kolumnę. Dzięki temu możemy zmienić organizację danych albo uzyskiwać różne rodzaje grup danych. Na przykład poniższe zapytanie zawiera klauzulę GROUP BY, w której dwukrotnie użyto kolumny division_id — raz w celu pogrupowania według division_id i ponownie w klauzuli ROLLUP:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id);

DIVI JOB_ SUM(SALARY)
-----
```

BIZ	KIE	530000
BIZ	PRA	280000
BIZ	PRE	800000
OPE	INŻ	245000
OPE	KIE	805000
OPE	PRA	270000
SPR	KIE	4446000
SPR	PRA	490000
WSP	KIE	465000
WSP	PRA	435000
WSP	TEC	115000
BIZ		1610000
OPE		1320000
SPR		4936000
WSP		1015000
BIZ		1610000
OPE		1320000
SPR		4936000
WSP		1015000

Należy jednak zauważyc, że ostatnie cztery wiersze są duplikatami wcześniejszych czterech wierszy. Możemy je usunąć, korzystając z funkcji `GROUP_ID()`, która zostanie opisana w kolejnym podrozdziale.

Użycie funkcji `GROUP_ID()`

Funkcję `GROUP_ID()` możemy zastosować do usunięcia powtarzających się wierszy zwroconych przez klawisz `GROUP BY`. Jeżeli dla określonego grupowania występuje n duplikatów, `GROUP_ID()` zwraca liczby w zakresie od 0 do $n - 1$.

W poniższym przykładzie zmieniono zapytanie przedstawione w poprzednim podrozdziale tak, aby zawierało wyniki z funkcji `GROUP_ID()`:

```
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id);
```

DIVI	JOB_	GROUP_ID()	SUM(SALARY)
BIZ	KIE	0	530000
BIZ	PRA	0	280000
BIZ	PRE	0	800000
OPE	INŻ	0	245000
OPE	KIE	0	805000
OPE	PRA	0	270000
SPR	KIE	0	4446000
SPR	PRA	0	490000
WSP	KIE	0	465000
WSP	PRA	0	435000
WSP	TEC	0	115000
BIZ		0	1610000
OPE		0	1320000
SPR		0	4936000
WSP		0	1015000
BIZ		1	1610000
OPE		1	1320000
SPR		1	4936000
WSP		1	1015000

Należy zauważyc, że ta funkcja zwraca 0 dla wszystkich wierszy oprócz ostatnich czterech, które są duplikatami wcześniejszych czterech wierszy (w tym przypadku funkcja `GROUP_ID()` zwraca 1).

Powtarzające się wiersze możemy usunąć za pomocą klawizuli `HAVING`, przepuszczającej jedynie te linie, w których `GROUP_ID()` zwraca 0. Na przykład:

```
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
```

```
GROUP BY division_id, ROLLUP(division_id, job_id)
HAVING GROUP_ID() = 0;

DIVI JOB_ GROUP_ID() SUM(SALARY)
-----
BIZ KIE      0    530000
BIZ PRA      0    280000
BIZ PRE      0    800000
OPE INŻ      0    245000
OPE KIE      0    805000
OPE PRA      0    270000
SPR KIE      0   4446000
SPR PRA      0    490000
WSP KIE      0    465000
WSP PRA      0    435000
WSP TEC      0    115000
BIZ          0   1610000
OPE          0   1320000
SPR          0   4936000
WSP          0   1015000
```

To kończy omówienie zaawansowanych klauzul GROUP BY.

Użycie CROSS APPLY i OUTER APPLY

Wprowadzone w Oracle Database 12c klauzule CROSS APPLY i OUTER APPLY służą do porównywania wierszy zwracanych przez dwa zapytania SELECT i zwracają pasujące do siebie wiersze w jednym połączonym zestawie wyników.

Przykłady z kolejnych podrozdziałów korzystają z tabel divisions i employees3. Poniższe zapytanie pobiera wiersze z tabeli divisions:

```
SELECT *
FROM divisions;
```

```
DIVI NAME
-----
SPR Sprzedaż
OPE Obsługa
WSP Wsparcie
BIZ Biznes
```

W tabeli employees3 znajdują się tylko pracownicy z działu Biznes. Pracownicy z tego działu mają oznaczenie BIZ w kolumnie division_id, co można zobaczyć w wynikach poniższego zapytania:

```
SELECT *
FROM employees3;
```

EMPLOYEE_ID	DIVI	JOB_	FIRST_NAME	LAST_NAME	SALARY
1	BIZ	PRE	Jan	Kowalski	800000
14	BIZ	KIE	Marek	Kowalski	155000
15	BIZ	KIE	Julia	Nowak	175000
19	BIZ	KIE	Tatiana	Zuch	200000
37	BIZ	PRA	Grzegorz	Nowak	280000

Nie ma tutaj pracowników działów Sprzedaż, Obsługa i Wsparcie.

CROSS APPLY

Klauzula CROSS APPLY pozwala uzyskać połączenie wierszy z dwóch instrukcji SELECT. W wynikach zwracane są tylko te wiersze z zewnętrznej instrukcji SELECT, które pasują do wewnętrznej instrukcji SELECT. Poniższy przykład pobiera wiersze z tabel divisions i employees3:

```
SELECT *
FROM divisions d
CROSS APPLY
  (SELECT *
   FROM employees3 e
   WHERE e.division_id = d.division_id)
ORDER BY employee_id;
```

DIVI	NAME	EMPLOYEE_ID	DIVI	JOB_	FIRST_NAME	LAST_NAME	SALARY
BIZ	Biznes	1	BIZ	PRE	Jan	Kowalski	800000
BIZ	Biznes	14	BIZ	KIE	Marek	Kowalski	155000
BIZ	Biznes	15	BIZ	KIE	Julia	Nowak	175000
BIZ	Biznes	19	BIZ	KIE	Tatiana	Zuch	200000
BIZ	Biznes	37	BIZ	PRA	Grzegorz	Nowak	280000

W wynikach łączone są informacje o jednostce biznesowej z tabeli `divisions` i pasujące do nich informacje o pracownikach z tabeli `employees3`.

OUTER APPLY

Klauzula `OUTER APPLY` zwraca połączenie wierszy z dwóch instrukcji `SELECT` włącznie z tymi wierszami zewnętrzną instrukcją `SELECT`, które nie mają połączenia z wierszami zwróconymi przez zewnętrzną instrukcję `SELECT`. Poniższy przykład pokazuje wiersze pobrane z tabel `divisions` i `employees3`:

```
SELECT *
FROM divisions d
OUTER APPLY
  (SELECT *
   FROM employees3 e
   WHERE e.division_id = d.division_id)
ORDER BY employee_id;
```

DIVI	NAME	EMPLOYEE_ID	DIVI	JOB_	FIRST_NAME	LAST_NAME	SALARY
BIZ	Biznes	1	BIZ	PRE	Jan	Kowalski	800000
BIZ	Biznes	14	BIZ	KIE	Marek	Kowalski	155000
BIZ	Biznes	15	BIZ	KIE	Julia	Nowak	175000
BIZ	Biznes	19	BIZ	KIE	Tatiana	Zuch	200000
BIZ	Biznes	37	BIZ	PRA	Grzegorz	Nowak	280000
WSP	Wsparcie						
SPR	Sprzedaż						
OPE	Obsługa						

W wynikach znajdują się pracownicy działu Biznes. Dodatkowo dołączone zostały wiersze dla pozostałych działów mimo tego, że nie ma należących do nich pracowników. Te dodatkowe wiersze są wyświetlane, ponieważ `OUTER APPLY` dołącza nie mające połączenia wiersze z zewnętrzną instrukcją `SELECT`. Kolumny, w których powinny znajdować się dane pracowników, zawierają wartości `NULL`.

LATERAL

Wprowadzona w Oracle Database 12c klauzula `LATERAL` umożliwia wykorzystanie podzapytania jako widoku wbudowanego. Widok wbudowany pobiera dane z jednej lub większej liczby tabel, tworząc tymczasową tabelę, którą zewnętrzne zapytanie może wykorzystać jako źródło danych. Poniższy przykład pokazuje wykorzystanie `LATERAL`:

```
SELECT *
FROM divisions d,
LATERAL
  (SELECT *
   FROM employees3 e
```

```

WHERE e.division_id = d.division_id
);
ORDER BY employee_id;

```

DIVI	NAME	EMPLOYEE_ID	DIVI	JOB_	FIRST_NAME	LAST_NAME	SALARY
BIZ	Biznes	1	BIZ	PRE	Jan	Kowalski	800000
BIZ	Biznes	14	BIZ	KIE	Marek	Kowalski	155000
BIZ	Biznes	15	BIZ	KIE	Julia	Nowak	175000
BIZ	Biznes	19	BIZ	KIE	Tatiana	Zuch	200000
BIZ	Biznes	37	BIZ	PRA	Grzegorz	Nowak	280000

Zwrócone zostały informacje na temat działu Biznes i jego pracowników.

LATERAL — w odróżnieniu od OUTER APPLY — nie tworzy pól zawierających wartość NULL dla niepasujących wierszy. Użycie LATERAL ma ograniczenia:

- LATERAL nie może być użyte w odwołaniu do tabeli z PIVOT i UNPIVOT.
- Widok wbudowany LATERAL nie może łączyć się lewostronnie z pierwszą tabelą w prawostronnym połączeniu zewnętrznym lub pełnym połączeniu zewnętrznym (ang. *right outer join, full outer join*).

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- operatory zestawu (UNION ALL, UNION, INTERSECT i MINUS) umożliwiają łączenie wierszy zwracanych przez kilka zapytań,
- funkcja TRANSLATE() konwertuje wystąpienia znaków w jednym napisie na znaki z drugiego napisu,
- wyrażenie CASE umożliwia wykorzystanie logiki warunkowej (*if-then-else*) w SQL,
- zapytania mogą dotyczyć danych hierarchicznych,
- ROLLUP rozszerza klauzulę GROUP BY o zwarcie wiersza zawierającego podsumowanie częściowe każdej grupy wierszy oraz wiersza zawierającego podsumowanie całkowite wszystkich grup,
- CUBE rozszerza klauzulę GROUP BY o zwarcie wierszy zawierających podsumowania częściowe dla wszystkich kombinacji kolumn oraz wiersza zawierającego podsumowanie całkowite,
- CROSS APPLY zwraca połączone wiersze wyników z dwóch instrukcji SELECT,
- OUTER APPLY zwraca połączone wiersze wyników z dwóch instrukcji SELECT razem z niepasującymi (niepołączonymi) wierszami zwróconymi przez zapytanie zewnętrzne,
- LATERAL zwraca wbudowany widok danych.

W kolejnym rozdziale zajmiemy się analizą danych.

ROZDZIAŁ

8

Analiza danych

Z tego rozdziału dowiesz się, jak:

- wykorzystywać funkcje analityczne, wykonujące złożone obliczenia,
- wykonywać obliczenia międzywierszowe za pomocą klauzuli **MODEL**,
- wykorzystywać klauzule **PIVOT** i **UNPIVOT**, które przydają się do przeglądania ogólnych trendów w dużych zbiorach danych,
- wykonywać zapytania zwracające pierwszych N lub ostatnich N wierszy wyników,
- odnajdywać wzorce w danych za pomocą klauzuli **MATCH_RECOGNIZE**.

Funkcje analityczne

Baza danych zawiera wiele wbudowanych funkcji analitycznych, umożliwiających wykonywanie złożonych obliczeń, takich jak wyszukanie najlepiej sprzedającego się rodzaju produktów w poszczególnych miesiącach, najlepszych sprzedawców itd. Funkcje analityczne możemy podzielić na następujące kategorie:

- **funkcje klasyfikujące** — umożliwiają obliczanie klasyfikacji, percentyl i n -tyli (tertyli, kwartyli itd.),
- **odwrotne funkcje percentylu** — umożliwiają obliczenie wartości odpowiadającej percentylowi,
- **funkcje okien** — umożliwiają obliczanie agregatów skumulowanych i ruchomych,
- **funkcje raportujące** — umożliwiają obliczanie na przykład udziałów w rynku,
- **funkcje LAG() i LEAD()** — umożliwiają pobranie wartości z wiersza, który jest oddalony o określona liczbę wierszy od bieżącego,
- **funkcje FIRST() i LAST()** — umożliwiają pobranie odpowiednio pierwszej i ostatniej wartości w uporządkowanej grupie,
- **funkcje regresji liniowej** — umożliwiające dopasowanie linii regresji metodą najmniejszych kwadratów do zestawu par liczb,
- **funkcje hipotetycznych klasyfikacji i dystrybucji** — umożliwiają obliczenie klasyfikacji i percentylu, w którym znalazły się nowy wiersz po wstawieniu go do tabeli.

Te funkcje zostaną opisane wkrótce, zaczniemy jednak od omówienia przykładowej tabeli.

Przykładowa tabela

W kolejnych podrozdziałach będziemy korzystali z tabeli `all_sales`. Składa się ona sumę wszystkich sprzedaży (w złotych) dla konkretnych lat, miesięcy, rodzajów produktu i pracownika. Tabela `all_sales` jest tworzona przez skrypt `store_schema.sql` za pomocą następującej instrukcji:

```

CREATE TABLE all_sales (
    year INTEGER NOT NULL,
    month INTEGER NOT NULL,
    prd_type_id INTEGER
        CONSTRAINT all_sales_fk_product_types
        REFERENCES product_types(product_type_id),
    emp_id INTEGER
        CONSTRAINT all_sales_fk_employees2
        REFERENCES employees2(employee_id),
    amount NUMBER(8, 2),
    CONSTRAINT all_sales_pk PRIMARY KEY (
        year, month, prd_type_id, emp_id
    )
);

```

Tabela `all_sales` zawiera pięć kolumn:

- `year`, w której składowany jest rok sprzedaży,
- `month`, w której składowany jest miesiąc sprzedaży (od 1 do 12),
- `prd_type_id`, w której składowany jest `product_type_id` produktu,
- `emp_id`, w której składowany jest `employee_id` pracownika obsługującego sprzedaż,
- `amount`, w której składowana jest wartość sprzedaży w złotych.

Poniższe zapytanie pobiera pierwsze 12 wierszy z tabeli `all_sales`:

```

SELECT *
FROM all_sales
WHERE ROWNUM <= 12;

```

YEAR	MONTH	PRD_TYPE_ID	EMP_ID	AMOUNT
2003	1	1	21	10034,84
2003	2	1	21	15144,65
2003	3	1	21	20137,83
2003	4	1	21	25057,45
2003	5	1	21	17214,56
2003	6	1	21	15564,64
2003	7	1	21	12654,84
2003	8	1	21	17434,82
2003	9	1	21	19854,57
2003	10	1	21	21754,19
2003	11	1	21	13029,73
2003	12	1	21	10034,84



Tabela `all_sales` zawiera znacznie więcej wierszy, ze względu jednak na ograniczoną ilość miejsca nie umieszczono całego listingu.

Przejdzmy do omówienia funkcji klasyfikujących.

Użycie funkcji klasyfikujących

Funkcje klasyfikujące służą do obliczania klasyfikacji, percentyl i n -tyli. Zostały przedstawione w tabeli 8.1. Zacznijmy od opisu funkcji `RANK()` i `DENSE_RANK()`.

Użycie funkcji `RANK()` i `DENSE_RANK()`

Funkcje `RANK()` i `DENSE_RANK()` służą do klasyfikowania elementów w grupie. Różnica między nimi dotyczy sposobu, w jaki zachowują się, gdy kilka elementów przypada na tę samą pozycję: funkcja `RANK()` pozostawia w takim przypadku lukę w sekwencji, a funkcja `DENSE_RANK()` nie pozostawia luk.

Na przykład gdybyśmy chcieli poklasyfikować sprzedaż według typów produktu, a dwa typy przypadałyby na pierwsze miejsce, funkcja `RANK()` umieściłaby dwa typy na pierwszym miejscu, ale kolejny typ

Tabela 8.1. Funkcje klasyfikujące

Funkcja	Opis
RANK()	Zwraca klasyfikację w grupie. Funkcja RANK() pozostawia luki w sekwencji rang w przypadku równowagi
DENSE_RANK()	Zwraca klasyfikację w grupie. Funkcja DENSE_RANK() nie pozostawia luki w sekwencji rang w przypadku równowagi
CUME_DIST()	Zwraca pozycję określonej wartości w grupie wartości. CUME_DIST() jest skrótem angielskiej nazwy dystrybutanty (ang. <i>cumulative distribution</i>)
PERCENT_RANK()	Zwraca procentową rangę wartości w grupie wartości
NTILE()	Zwraca n -tyle: tertyle, kwartyle itd.
ROW_NUMBER()	Zwraca numer z każdym wierszem z grupy

znajdowałby się na trzeciej pozycji. Funkcja DENSE_RANK() natomiast również umieściłaby dwa rodzaje produktów na pierwszym miejscu, kolejny znajdowałby się jednak na drugiej pozycji.

Poniższe zapytanie obrazuje zastosowanie funkcji RANK() i DENSE_RANK() do pobrania klasyfikacji sprzedaży według rodzajów produktu w 2003 roku. Należy zwrócić uwagę na użycie słowa kluczowego OVER w wypołanach funkcji RANK() i DENSE_RANK():

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081,84	1	1
2	186381,22	4	4
3	478270,91	2	2
4	402751,16	3	3

Sprzedaż produktu nr 1 jest klasyfikowana jako 1, sprzedaż produktu nr 2 — jako 4 itd. Ponieważ nie występują konflikty, funkcje RANK() i DENSE_RANK() zwracają takie same rangi.

Kolumna amount tabeli all_sales zawiera wartość NULL dla wszystkich wierszy, w których PRD_TYPE_ID wynosi 5. W poprzednim zapytaniu te wiersze zostały pominięte na skutek umieszczenia w klauzuli WHERE warunku AND amount IS NOT NULL. W kolejnym przykładzie zostały one dołączone w wyniku pominięcia wiersza AND z klauzuli WHERE:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081,84	2	2
2	186381,22	5	5
3	478270,91	3	3
4	402751,16	4	4
5		1	1

Ostatni wiersz zawiera wartość NULL jako sumę w kolumnie `amount` i w tym wierszu funkcje `RANK()` i `DENSE_RANK()` zwracają 1. Jest tak dlatego, że domyślnie w klasyfikacjach malejących obydwie te funkcje przypisują wartościom NULL najwyższą rangę (1) (jeżeli słowo kluczowe `DESC` zostanie użyte w klauzuli `OVER`) i najniższą rangę w rankingach rosnących (jeżeli w klauzuli `OVER` zostanie użyte słowo kluczowe `ASC`).

Sterowanie klasyfikowaniem wartości NULL

za pomocą klauzul `NULLS FIRST` i `NULLS LAST`

W funkcjach analitycznych za pomocą klauzul `NULLS FIRST` i `NULLS LAST` możemy jawnie określić, czy wartości NULL mają być najwyższe, czy też najniższe w grupie. W poniższym przykładzie użyto klauzuli `NULLS LAST` do określenia, że wartości NULL są wartościami najniższymi:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS dense_rank
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081,84	1	1
2	186381,22	4	4
3	478270,91	2	2
4	402751,16	3	3
5		5	5

Użycie klauzuli `PARTITION BY` w funkcjach analitycznych

Klauzulę `PARTITION BY` stosujemy w funkcjach analitycznych, jeżeli musimy podzielić grupy na podgrupy. Na przykład jeżeli chcemy podzielić wartości sprzedaży na miesiące, możemy użyć klauzuli `PARTITION BY month`, tak jak w poniższym zapytaniu:

```
SELECT
    prd_type_id, month, SUM(amount),
    RANK() OVER (PARTITION BY month ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id, month
ORDER BY prd_type_id, month;
```

PRD_TYPE_ID	MONTH	SUM(AMOUNT)	RANK
1	1	38909,04	1
1	2	70567,9	1
1	3	91826,98	1
1	4	120344,7	1
1	5	97287,36	1
1	6	57387,84	1
1	7	60929,04	2
1	8	75608,92	1
1	9	85027,42	1
1	10	105305,22	1
1	11	55678,38	1
1	12	46209,04	2
2	1	14309,04	4
2	2	13367,9	4
2	3	16826,98	4
2	4	15664,7	4
2	5	18287,36	4

2	6	14587,84	4
2	7	15689,04	3
2	8	16308,92	4
2	9	19127,42	4
2	10	13525,14	4
2	11	16177,84	4
2	12	12509,04	4
3	1	24909,04	2
3	2	15467,9	3
3	3	20626,98	3
3	4	23844,7	2
3	5	18687,36	3
3	6	19887,84	3
3	7	81589,04	1
3	8	62408,92	2
3	9	46127,42	3
3	10	70325,29	3
3	11	46187,38	2
3	12	48209,04	1
4	1	17398,43	3
4	2	17267,9	2
4	3	31026,98	2
4	4	16144,7	3
4	5	20087,36	2
4	6	33087,84	2
4	7	12089,04	4
4	8	58408,92	3
4	9	49327,42	2
4	10	75325,14	2
4	11	42178,38	3
4	12	30409,05	3

Użycie operatorów ROLLUP, CUBE i GROUPING SETS w funkcjach analitycznych

Z funkcjami analitycznymi mogą być stosowane operatory ROLLUP, CUBE i GROUPING SETS. W poniższym zapytaniu użyto operatora ROLLUP oraz funkcji RANK(), aby uzyskać klasyfikację wartości sprzedaży według identyfikatorów typów produktu:

```
SELECT
  prd_type_id, SUM(amount),
  RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY ROLLUP(prd_type_id)
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK
1	905081,84	3
2	186381,22	6
3	478270,91	4
4	402751,16	5
5		1
	1972485,13	2

W kolejnym zapytaniu użyto CUBE i RANK() do uzyskania całej klasyfikacji wartości sprzedaży według identyfikatorów typu produktu oraz identyfikatorów pracowników:

```
SELECT
  prd_type_id, emp_id, SUM(amount),
  RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY CUBE(prd_type_id, emp_id)
ORDER BY prd_type_id, emp_id;
```

210 Oracle Database 12c i SQL. Programowanie

PRD_TYPE_ID	EMP_ID	SUM(AMOUNT)	RANK
1	21	197916,96	19
1	22	214216,96	17
1	23	98896,96	26
1	24	207216,96	18
1	25	93416,96	28
1	26	93417,04	27
1		905081,84	9
2	21	20426,96	40
2	22	19826,96	41
2	23	19726,96	42
2	24	43866,96	34
2	25	32266,96	38
2	26	50266,42	31
2		186381,22	21
3	21	140326,96	22
3	22	116826,96	23
3	23	112026,96	24
3	24	34829,96	36
3	25	29129,96	39
3	26	45130,11	33
3		478270,91	10
4	21	108326,96	25
4	22	81426,96	30
4	23	92426,96	29
4	24	47456,96	32
4	25	33156,96	37
4	26	39956,36	35
4		402751,16	13
5	21		1
5	22		1
5	23		1
5	24		1
5	25		1
5	26		1
5			1
	21	466997,84	11
	22	432297,84	12
	23	323077,84	15
	24	333370,84	14
	25	187970,84	20
	26	228769,93	16
		1972485,13	8

W kolejnym zapytaniu użyto GROUPING SETS i RANK() do pobrania jedynie klasyfikacji częściowych podsumowań wartości sprzedazy:

```
SELECT
    prd_type_id, emp_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY GROUPING SETS(prd_type_id, emp_id)
ORDER BY prd_type_id, emp_id;
```

PRD_TYPE_ID	EMP_ID	SUM(AMOUNT)	RANK
1		905081,84	2
2		186381,22	11
3		478270,91	3
4		402751,16	6
5			1
	21	466997,84	4
	22	432297,84	5

23	323077,84	8
24	333370,84	7
25	187970,84	10
26	228769,93	9

Użycie funkcji CUME_DIST() i PERCENT_RANK()

Funkcja CUME_DIST() służy do wyznaczania pozycji określonej wartości w grupie, a funkcja PERCENT_RANK() — do wyznaczania rangi procentowej określonej wartości względem grupy wartości.

Poniższe zapytanie obrazuje użycie funkcji CUME_DIST() i PERCENT_DIST() do pobrania dystrybuanty i klasyfikacji procentowej wartości sprzedaży:

```
SELECT
    prd_type_id, SUM(amount),
    CUME_DIST() OVER (ORDER BY SUM(amount) DESC) AS cume_dist,
    PERCENT_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent_rank
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	CUME_DIST	PERCENT_RANK
1	905081,84	,4	,25
2	186381,22	1	1
3	478270,91	,6	,5
4	402751,16	,8	,75
5	,	,2	0

Użycie funkcji NTILE()

Funkcja NTILE(*kubelki*) służy do obliczania *n*-tyli (tertyli, kwartyli itd.). Parametr *kubelki* określa liczbę „kubelków”, w których zostaną porozmieszczone grupy wierszy. Na przykład:

- NTILE(2) określa dwa kubelki, w związku z czym wiersze zostaną podzielone na dwie grupy wierszy,
- NTILE(4) dzieli grupy na cztery kubelki, w wyniku czego dzieli wiersze na cztery grupy.

Poniższe zapytanie obrazuje użycie funkcji NTILE(). Przesłano do niej wartość 4, aby podzielić grupy wierszy na cztery kubelki:

```
SELECT
    prd_type_id, SUM(amount),
    NTILE(4) OVER (ORDER BY SUM(amount) DESC) AS ntile
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	NTILE
1	905081,84	1
2	186381,22	4
3	478270,91	2
4	402751,16	3

Użycie funkcji ROW_NUMBER()

Funkcja ROW_NUMBER() zwraca liczbę (od 1) z każdym wierszem w grupie. Poniższe zapytanie obrazuje jej użycie:

```
SELECT
    prd_type_id, SUM(amount),
    ROW_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS row_number
```

```

FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;

PRD_TYPE_ID SUM(AMOUNT) ROW_NUMBER
-----
1    905081,84      2
2    186381,22      5
3    478270,91      3
4    402751,16      4
5    1

```

Na tym zakończymy omówienie funkcji rankingowych.

Użycie odwrotnych funkcji rankingowych

Z podrozdziału „Użycie funkcji CUME_DIST() i PERCENT_RANK()” dowiedziałeś się, że funkcja CUME_DIST() służy do obliczania pozycji określonej wartości w grupie wartości. Wiesz też, że za pomocą funkcji PERCENT_RANK() oblicza się rangę procentową wartości w grupie wartości.

Ten podrozdział dotyczy używania odwrotnych funkcji rankingowych działających odwrotnie do funkcji CUME_DIST() i PERCENT_RANK(). Dostępne są dwie odwrotne funkcje rankingowe:

- PERCENTILE_DISC(x) bada wartości dystrybuanty w grupie, aż odnajdzie taką, która jest większa od x lub równa mu,
- PERCENTILE_CONT(x) bada wartości rankingu procentowego aż do wyszukania takiej, która jest większa od x lub równa mu.

W poniższym zapytaniu użyto funkcji PERCENTILE_CONT() i PERCENTILE_DISC() do pobrania sumy wartości sprzedaży, dla której percentyl jest większy od 0,6 lub równy tej wartości:

```

SELECT
  PERCENTILE_CONT(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS percentile_cont,
  PERCENTILE_DISC(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS percentile_disc
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id;

PERCENTILE_CONT PERCENTILE_DISC
-----
417855,11      402751,16

```

Jeżeli porównamy powyższe sumy z wynikami zwróconymi w podrozdziale „Użycie funkcji CUME_DIST() i PERCENT_RANK()”, zobaczymy, że odpowiadają one tym wartośćom, dla których dystrybuanta i ranga procentowa wynoszą odpowiednio 0,6 i 0,75.

Użycie funkcji okna

Funkcje okna służą do obliczania sum kumulacyjnych i średnich kroczących w określonych zakresach wierszy, zakresie wartości czy przedziale czasu.

Zapytanie zwraca zestaw wierszy zwany zestawem wyników. Termin „okno” oznacza podzbiór wierszy zestawu wyników. Ten podzbiór „widziany” przez okno jest przetwarzany przez funkcje okna, które zwracają wartość. Możemy zdefiniować początek i koniec okna.

Okno może być użyte z następującymi funkcjami: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE() i STDDEV(), które zostały opisane w rozdziale 4. Funkcje okna mogą być również wykorzystywane z funkcjami FIRST_VALUE(), LAST_VALUE() oraz NTH_VALUE(), które zwracają pierwszą, ostatnią i n -tą wartość w oknie.Więcej informacji na temat tych funkcji znajduje się w dalszej części tego rozdziału. Z tego podrozdziału dowiesz się, jak obliczyć sumę kumulacyjną, średnią kroczącą i średnią centralną.

Obliczanie sumy kumulacyjnej

Poniższe zapytanie oblicza sumę kumulacyjną wartości sprzedaży w 2003 roku — od stycznia do grudnia. Należy zauważyć, że wartość sprzedaży w każdym miesiącu jest dodawana do wartości kumulacyjnej, która zwiększa się co miesiąc:

```
SELECT
    month, SUM(amount) AS month_amount,
    SUM(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS cumulative_amount
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

	MONTH	MONTH_AMOUNT	CUMULATIVE_AMOUNT
1	95525,55	95525,55	
2	116671,6	212197,15	
3	160307,92	372505,07	
4	175998,8	548503,87	
5	154349,44	702853,31	
6	124951,36	827804,67	
7	170296,16	998100,83	
8	212735,68	1210836,51	
9	199609,68	1410446,19	
10	264480,79	1674926,98	
11	160221,98	1835148,96	
12	137336,17	1972485,13	

W tym zapytaniu do obliczenia agregatu kumulacyjnego jest wykorzystywane następujące wyrażenie:

```
SUM(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS cumulative_amount
```

Omówimy teraz poszczególne części tego wyrażenia:

- **SUM(amount)** oblicza sumę wartości sprzedaży. Zewnętrzna funkcja **SUM()** oblicza wartość skumulowaną.
- **ORDER BY month** porządkuje według miesięcy wiersze odczytywane przez zapytanie.
- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** definiuje początek i koniec okna. Początek jest ustawiany przez **UNBOUNDED PRECEDING**, co oznacza, że początek okna jest ustawiony na stałe jako pierwszy wiersz w zestawie wyników zwróconym przez zapytanie. Koniec okna jest ustawiany jako **CURRENT ROW**, czyli bieżący wiersz z przetwarzanego zestawu wyników. Koniec okna zsuwa się o jeden wiersz w dół po wykonaniu obliczeń przez zewnętrzną funkcję **SUM()** i zwraca bieżącą wartość skumulowaną.

Całe zapytanie oblicza i zwraca sumę kumulacyjną wartości sprzedaży, rozpoczynając od pierwszego miesiąca i dodając wartość sprzedaży w drugim miesiącu, później w trzecim itd. aż do ostatniego miesiąca włącznie. Początek okna jest ustawiony na stałe w pierwszym miesiącu, ale okno przesuwa się w dół (po jednym wierszu zestawu wyników) po dodaniu wartości sprzedaży w danym miesiącu do sumy skumulowanej. Ten proces trwa do czasu przetworzenia przez okno i funkcje **SUM()** ostatniego wiersza zestawu wyników.

Nie należy pomylić końca okna z końcem zestawu wyników. W poprzednim przykładzie koniec okna przesuwał się w dół o jeden wiersz w zestawie wyników po przetworzeniu danego wiersza (czyli po dodaniu sumy wartości sprzedaży w danym miesiącu do sumy kumulacyjnej). W tym przykładzie koniec okna na początku znajduje się w pierwszym wierszu, wartość sumy sprzedaży w tym miesiącu jest dodawana do sumy skumulowanej, a następnie koniec okna przesuwa się w dół o jeden wiersz — do drugiego wiersza. W tym momencie okno „widzi” dwa wiersze. Suma wartości sprzedaży w tym miesiącu jest

dodawana do sumy skumulowanej i koniec okna przesuwa się w dół o jeden wiersz, do trzeciego wiersza. W tym momencie okno „widzi” trzy wiersze. Ten proces trwa aż do osiągnięcia dwunastego wiersza. Wówczas okno „widzi” dwanaście wierszy.

W poniższym przykładzie zapytanie oblicza skumulowaną wartość sprzedaży, rozpoczynając od czerwca 2003 roku (szóstego miesiąca) i kończąc w grudniu 2003 (na dwunastym miesiącu):

```

SELECT
    month, SUM(amount) AS month_amount,
    SUM(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
        cumulative_amount
FROM all_sales
WHERE year = 2003
AND month BETWEEN 6 AND 12
GROUP BY month
ORDER BY month;

MONTH MONTH_AMOUNT CUMULATIVE_AMOUNT
-----
6      124951,36   124951,36
7      170296,16   295247,52
8      212735,68   507983,2
9      199609,68   707592,88
10     264480,79   972073,67
11     160221,98   1132295,65
12     137336,17   1269631,82

```

Obliczanie średniej kroczącej

Poniższe zapytanie oblicza średnią kroczącą wartości sprzedaży między bieżącym miesiącem i poprzednimi trzema:

```

SELECT
    month, SUM(amount) AS month_amount,
    AVG(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
        AS moving_average
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;

MONTH MONTH_AMOUNT MOVING_AVERAGE
-----
1      95525,55   95525,55
2      116671,6   106098,575
3      160307,92  124168,357
4      175998,8   137125,968
5      154349,44  151831,94
6      124951,36  153901,88
7      170296,16  156398,94
8      212735,68  165583,16
9      199609,68  176898,22
10     264480,79  211780,578
11     160221,98  209262,033
12     137336,17  190412,155

```

Należy zauważyć, że do obliczenia średniej kroczącej jest używane poniższe wyrażenie:

```

AVG(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS moving_average

```

Przeanalizujmy poszczególne części tego wyrażenia:

- **SUM(amount)** oblicza sumę wartości sprzedaży. Zewnętrzna funkcja **AVG()** oblicza średnią.
- **ORDER BY month** porządkuje według miesięcy wiersze odczytywane przez zapytanie.
- **ROWS BETWEEN 3 PRECEDING AND CURRENT ROW** definiuje początek okna — obejmuje on trzy wiersze poprzedzające bieżący wiersz. Końcem okna jest aktualnie przetwarzany wiersz.

Całe wyrażenie oblicza więc średnią kroczącą wartości sprzedaży między bieżącym miesiącem i poprzednimi trzema. Ponieważ w przypadku pierwszych dwóch miesięcy dostępne są dane z mniej niż trzech miesięcy, średnia krocząca jest obliczana jedynie na podstawie dostępnych informacji.

Zarówno początek, jak i koniec okna na samym początku znajdują się w pierwszym wierszu pobranym przez zapytanie. Koniec okna przesuwa się w dół po przetworzeniu każdego wiersza. Początek okna przesuwa się w dół dopiero po przetworzeniu czwartego wiersza i dalej przesuwa się w dół o jeden wiersz po przetworzeniu każdego wiersza. Ten proces trwa aż do przetworzenia ostatniego wiersza z zestawu wyników.

Obliczanie średniej centralnej

Poniższe zapytanie oblicza średnią kroczącą wartości sprzedaży między bieżącym, poprzednim i następnym miesiącem:

```
SELECT
    month, SUM(amount) AS month_amount,
    AVG(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS moving_average
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	MOVING_AVERAGE
1	95525,55	106098,575
2	116671,6	124168,357
3	160307,92	150992,773
4	175998,8	163552,053
5	154349,44	151766,533
6	124951,36	149865,653
7	170296,16	169327,733
8	212735,68	194213,84
9	199609,68	225608,717
10	264480,79	208104,15
11	160221,98	187346,313
12	137336,17	148779,075

Do obliczenia średniej kroczącej w powyższym zapytaniu użyto następującego wyrażenia:

```
AVG(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS moving_average
```

Składa się ono z następujących części:

- **SUM(amount)** oblicza sumę wartości sprzedaży. Zewnętrzna funkcja **AVG()** oblicza średnią.
- **ORDER BY month** porządkuje według miesięcy wiersze pobrane przez zapytanie.
- **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING** definiuje początek okna jako zawierający wiersz poprzedzający bieżący. Koniec okna znajduje się w wierszu następującym po bieżącym.

Wyrażenie oblicza średnią kroczącą wartości sprzedaży dla bieżącego, poprzedniego i następnego miesiąca. Ponieważ w przypadku pierwszego i ostatniego miesiąca nie dysponujemy danymi dla pełnych trzech miesięcy, średnia krocząca jest obliczana jedynie na podstawie dostępnych informacji.

Początek okna znajduje się najpierw w pierwszym wierszu, odczytanym przez zapytanie. Koniec okna znajduje się wówczas w drugim wierszu i przesuwa się w dół po przetworzeniu każdego wiersza. Początek okna zaczyna przesuwać się w dół po przetworzeniu drugiego wiersza. Przetwarzanie kończy się w ostatnim wierszu odczytanym przez zapytanie.

Pobieranie pierwszego i ostatniego wiersza za pomocą funkcji FIRST_VALUE() i LAST_VALUE()

Funkcje FIRST_VALUE() i LAST_VALUE() służą do pobierania pierwszego i ostatniego wiersza z okna. W poniższym zapytaniu użyto funkcji FIRST_VALUE i LAST_VALUE do pobrania wartości sprzedaży w poprzednim i kolejnym miesiącu:

```
SELECT
    month, SUM(amount) AS month_amount,
    FIRST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS previous_month_amount,
    LAST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS next_month_amount
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

	MONTH	MONTH_AMOUNT	PREVIOUS_MONTH_AMOUNT	NEXT_MONTH_AMOUNT
1	95525,55	95525,55	116671,6	
2	116671,6	95525,55	160307,92	
3	160307,92	116671,6	175998,8	
4	175998,8	160307,92	154349,44	
5	154349,44	175998,8	124951,36	
6	124951,36	154349,44	170296,16	
7	170296,16	124951,36	212735,68	
8	212735,68	170296,16	199609,68	
9	199609,68	212735,68	264480,79	
10	264480,79	199609,68	160221,98	
11	160221,98	264480,79	137336,17	
12	137336,17	160221,98	137336,17	

W kolejnym zapytaniu wartość sprzedaży w bieżącym miesiącu jest dzielona przez wartość sprzedaży w poprzednim miesiącu (etykieta curr_div_prev) oraz przez wartość sprzedaży w następnym miesiącu (etykieta curr_div_next):

```
SELECT
    month, SUM(amount) AS month_amount,
    SUM(amount)/FIRST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS curr_div_prev,
    SUM(amount)/LAST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS curr_div_next
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

	MONTH	MONTH_AMOUNT	CURR_DIV_PREV	CURR_DIV_NEXT
1	95525,55	1	,818755807	
2	116671,6	1,22136538	,727796855	
3	160307,92	1,37400978	,910846665	
4	175998,8	1,09787963	1,14026199	

5	154349,44	,876991434	1,23527619
6	124951,36	,809535558	,733729756
7	170296,16	1,36289961	,800505867
8	212735,68	1,24921008	1,06575833
9	199609,68	,93829902	,754722791
10	264480,79	1,3249898	1,65071478
11	160221,98	,605798175	1,16664081
12	137336,17	,857161858	1

Pobieranie n-tego wiersza za pomocą funkcji NTH_VALUE()

Funkcja NTH_VALUE() zwraca n -ty wiersz z okna. Ta funkcja została wprowadzona w Oracle Database 11g Release 2. Poniższe zapytanie wykorzystuje NTH_VALUE() do pobrania wartości sprzedaży w drugim miesiącu analizowanego okna w konstrukcji NTH_VALUE(SUM(amount), 2):

```
SELECT
    month, SUM(amount) AS month_amount,
    NTH_VALUE(SUM(amount), 2) OVER (
        ORDER BY month ROWS BETWEEN
        UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) nth_value
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	NTH_VALUE
1	95525,55	116671,6
2	116671,6	116671,6
3	160307,92	116671,6
4	175998,8	116671,6
5	154349,44	116671,6
6	124951,36	116671,6
7	170296,16	116671,6
8	212735,68	116671,6
9	199609,68	116671,6
10	264480,79	116671,6
11	160221,98	116671,6
12	137336,17	116671,6

Kolejne zapytanie wykorzystuje NTH_VALUE() do pobrania maksymalnej sprzedaży pracownika nr 24 dla produktów typu 1, 2 i 3. Wartość ta znajduje się na 4. pozycji w oknie i jest pobierana za pomocą wyrażenia NTH_VALUE(MAX(amount), 4).

```
SELECT
    prd_type_id, emp_id, MAX(amount),
    NTH_VALUE(MAX(amount), 4) OVER (
        PARTITION BY prd_type_id ORDER BY emp_id
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) nth_value
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 3
GROUP BY prd_type_id, emp_id
ORDER BY prd_type_id, emp_id;
```

PRD_TYPE_ID	EMP_ID	MAX(AMOUNT)	NTH_VALUE
1	21	25057,45	25214,56
1	22	29057,45	25214,56
1	23	16057,45	25214,56
1	24	25214,56	25214,56
1	25	14057,45	25214,56
1	26	16754,27	25214,56
2	21	2754,19	7314,56

2	22	2657,45	7314,56
2	23	2357,45	7314,56
2	24	7314,56	7314,56
2	25	5364,84	7314,56
2	26	5434,84	7314,56
3	21	32754,19	6337,83
3	22	27264,84	6337,83
3	23	23264,84	6337,83
3	24	6337,83	6337,83
3	25	4364,64	6337,83
3	26	5457,45	6337,83

Na tym zakończymy omówienie funkcji okna.

Funkcje raportujące

Funkcje raportujące służą do wykonywania obliczeń uwzględniających różne grupy i partycje wewnętrz nich.

Raportowanie może być wykonywane z użyciem następujących funkcji: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE() i STDEV(). Można ponadto użyć funkcji RATIO_TO_REPORT() do obliczenia stosunku wartości do sumy wartości z zestawu oraz funkcji LISTAGG() do uporządkowania wierszy w grupie i połączenia zestawu grupowanych wartości.

Z tego podrozdziału dowiesz się, jak raportować sumy i stosować funkcje RATIO_TO_REPORT() oraz LISTAGG().

Raportowanie sumy

Poniższe zapytanie raportuje dla pierwszych trzech miesięcy 2003 roku następujące elementy:

- całkowitą sumę wszystkich wartości sprzedaży we wszystkich trzech miesiącach (etykieta total_month_amount),
- całkowitą sumę wszystkich wartości sprzedaży dla wszystkich typów produktów (etykieta total_product_type_amount).

```
SELECT
    month, prd_type_id,
    SUM(SUM(amount)) OVER (PARTITION BY month)
        AS total_month_amount,
    SUM(SUM(amount)) OVER (PARTITION BY prd_type_id)
        AS total_product_type_amount
FROM all_sales
WHERE year = 2003
AND month <= 3
GROUP BY month, prd_type_id
ORDER BY month, prd_type_id;
```

MONTH	PRD_TYPE_ID	TOTAL_MONTH_AMOUNT	TOTAL_PRODUCT_TYPE_AMOUNT
1	1	95525,55	201303,92
1	2	95525,55	44503,92
1	3	95525,55	61003,92
1	4	95525,55	65693,31
1	5	95525,55	
2	1	116671,6	201303,92
2	2	116671,6	44503,92
2	3	116671,6	61003,92
2	4	116671,6	65693,31
2	5	116671,6	
3	1	160307,92	201303,92
3	2	160307,92	44503,92
3	3	160307,92	61003,92
3	4	160307,92	65693,31
3	5	160307,92	

Do raportowania całkowitej sumy sprzedaży we wszystkich trzech miesiącach (etykieta `total_month_amount`) użyto wyrażenia:

```
SUM(SUM(amount)) OVER (PARTITION BY month)
AS total_month_amount
```

Składa się ono z następujących części:

- `SUM(amount)` oblicza średnią wartość sprzedaży. Zewnętrzna funkcja `SUM()` oblicza sumę całkowitą.
- `OVER (PARTITION BY month)` powoduje, że zewnętrzna funkcja `SUM()` oblicza sumę dla każdego miesiąca.

Do raportowania całkowitej sumy sprzedaży wszystkich rodzajów produktów (etykieta `total_product_type_amount`) w powyższym zapytaniu użyto następującego wyrażenia:

```
SUM(SUM(amount)) OVER (PARTITION BY prd_type_id)
AS total_product_type_amount
```

Składają się na nie następujące elementy:

- `SUM(amount)` oblicza średnią wartość sprzedaży. Zewnętrzna funkcja `SUM()` oblicza sumę całkowitą.
- `OVER (PARTITION BY prd_type_id)` powoduje, że zewnętrzna funkcja `SUM()` oblicza sumę dla każdego typu produktu.

Użycie funkcji RATIO_TO_REPORT()

Funkcja `RATIO_TO_REPORT()` służy do obliczania stosunku wartości do sumy zestawu wartości.

Poniższe zapytanie raportuje dla pierwszych trzech miesięcy 2003 roku następujące informacje:

- sumę wartości sprzedaży według typów produktu w każdym miesiącu (etykieta `prd_type_amount`),
- stosunek wartości sprzedaży danego rodzaju produktu do całkowitej sprzedaży w miesiącu (etykieta `prd_type_ratio`). Ta wartość jest obliczana za pomocą funkcji `RATIO_TO_REPORT()`.

```
SELECT
    month, prd_type_id,
    SUM(amount) AS prd_type_amount,
    RATIO_TO_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd_type_ratio
FROM all_sales
WHERE year = 2003
AND month <= 3
GROUP BY month, prd_type_id
ORDER BY month, prd_type_id;
```

MONTH	PRD_TYPE_ID	PRD_TYPE_AMOUNT	PRD_TYPE_RATIO
1	1	38909,04	,40731553
1	2	14309,04	,149792804
1	3	24909,04	,260757881
1	4	17398,43	,182133785
1	5		
2	1	70567,9	,604842138
2	2	13367,9	,114577155
2	3	15467,9	,132576394
2	4	17267,9	,148004313
2	5		
3	1	91826,98	,57281624
3	2	16826,98	,104966617
3	3	20626,98	,128670998
3	4	31026,98	,193546145
3	5		

Do obliczenia wspomnianego stosunku (etykieta `prd_type_ratio`) użyto następującego wyrażenia:

```
RATIO_TO_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd_type_ratio
```

Jego składowe to:

- `SUM(amount)` oblicza sumę wartości sprzedaży.
- `OVER (PARTITION BY month)` powoduje, że zewnętrzna funkcja `SUM()` oblicza sumę wartości sprzedaży w każdym miesiącu,
- stosunek jest obliczany przez podzielenie sumy wartości sprzedaży poszczególnych typów produktów przez całkowitą wartość (sumę) sprzedaży w danym miesiącu.

Użycie funkcji LISTAGG()

Funkcja `LISTAGG()` porządkuje wiersze w grupie i łączy zgrupowane wartości. Ta funkcja została wprowadzona w Oracle Database 11g Release 2. Poniższe zapytanie pobiera produkty od 1 do 5 z tabeli `products` uporządkowane według ceny i nazwy, a następnie zwraca produkty najdroższe:

```
SELECT
  LISTAGG(name, ', ') WITHIN GROUP (ORDER BY price, name) AS "Lista produktów",
  MAX(price) AS "Najdroższy"
FROM products
WHERE product_id <= 5;
```

Lista produktów	Najdroższy
Wojny czołgów, Nauka współczesna, Supernowa, Chemia, Z Files	49,99

Następne zapytanie pobiera produkty od 1 do 5 z tabeli `products` i dla każdego produktu używa `LISTAGG()`, by wyświetlić produkty z taką samą wartością `product_type_id`:

```
SELECT
  product_id, product_type_id, name,
  LISTAGG(name, ', ')
    WITHIN GROUP (ORDER BY name)
    OVER (PARTITION BY product_type_id) AS "Product List"
FROM products
WHERE product_id <= 5
ORDER BY product_id, product_type_id;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	Product List
1	1	Nauka współczesna	Chemia, Nauka współczesna
2	1	Chemia	Chemia, Nauka współczesna
3	2	Supernowa	Supernowa, Wojny czołgów, Z Files
4	2	Wojny czołgów	Supernowa, Wojny czołgów, Z Files
5	2	Z Files	Supernowa, Wojny czołgów, Z Files

Na tym zakończymy omówienie funkcji raportujących.

Użycie funkcji LAG() i LEAD()

Funkcje `LAG()` i `LEAD()` służą do pobierania wartości z wiersza, który jest oddalony o określoną liczbę wierszy od bieżącego. W poniższym zapytaniu użyto tych funkcji do pobrania wartości sprzedaży w poprzednim i kolejnym miesiącu w stosunku do bieżącego:

```
SELECT
  month, SUM(amount) AS month_amount,
  LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous_month_amount,
  LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next_month_amount
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	PREVIOUS_MONTH_AMOUNT	NEXT_MONTH_AMOUNT
1	95525,55		116671,6
2	116671,6	95525,55	160307,92
3	160307,92	116671,6	175998,8

4	175998,8	160307,92	154349,44
5	154349,44	175998,8	124951,36
6	124951,36	154349,44	170296,16
7	170296,16	124951,36	212735,68
8	212735,68	170296,16	199609,68
9	199609,68	212735,68	264480,79
10	264480,79	199609,68	160221,98
11	160221,98	264480,79	137336,17
12	137336,17	160221,98	

Wartość sprzedaży w tych okresach jest pobierana za pomocą następujących wyrażeń:

`LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous_month_amount,`
`LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next_month_amount`

`LAG(SUM(amount), 1)` pobiera sumę wartości z poprzedniego wiersza, a `LEAD(SUM(amount), 1)` — z następnego.

Użycie funkcji FIRST i LAST

Funkcje FIRST i LAST pobierają pierwszą i ostatnią wartość w uporządkowanej grupie. Mogą zostać użyte z następującymi funkcjami: `SUM()`, `AVG()`, `MAX()`, `MIN()`, `COUNT()`, `STDDEV()` i `VARIANCE()`.

W poniższym zapytaniu użyto funkcji FIRST i LAST do pobrania danych o tym, w których miesiącach 2003 roku sprzedaż była najwyższa i najniższa:

```
SELECT
    MIN(month) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount))
    AS highest_sales_month,
    MIN(month) KEEP (DENSE_RANK LAST ORDER BY SUM(amount))
    AS lowest_sales_month
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;

HIGHEST_SALES_MONTH LOWEST_SALES_MONTH
----- -----
          1             10
```

Użycie funkcji regresji liniowej

Funkcje regresji liniowej służą do dopasowania metodą najmniejszych kwadratów linii regresji do zestawu par liczb. Mogą być używane jako funkcje agregujące, funkcje okien lub raportujące.

Funkcje regresji liniowej zostały opisane w tabeli 8.2. W składni funkcji y jest interpretowane przez funkcje jak zmienna zależna od x .

Tabela 8.2. Funkcje regresji liniowej

Funkcja	Opis
<code>REGR_AVGX(y, x)</code>	Zwraca średnią x po usunięciu par x i y , w których jedna (lub obie) wartości to NULL
<code>REGR_AVGY(y, x)</code>	Zwraca średnią y po usunięciu par x i y , w których jedna (lub obie) wartości to NULL
<code>REGR_COUNT(y, x)</code>	Zwraca liczbę par liczbowych, nierównych NULL, które są używane do dopasowania linii regresji
<code>REGR_INTERCEPT(y, x)</code>	Zwraca punkt przecięcia się linii regresji z osią Y
<code>REGR_R2(y, x)</code>	Zwraca współczynnik determinacji (R kwadrat) linii regresji
<code>REGR_SLOPE(y, x)</code>	Zwraca nachylenie linii regresji
<code>REGR_SXX(y, x)</code>	Zwraca $\text{REG_COUNT}(y, x) * \text{VAR_POP}(x)$
<code>REGR_SXY(y, x)</code>	Zwraca $\text{REG_COUNT}(y, x) * \text{COVAR_POP}(y, x)$
<code>REGR_SYY(y, x)</code>	Zwraca $\text{REG_COUNT}(y, x) * \text{VAR_POP}(y)$

Poniższe zapytanie obrazuje użycie funkcji regresji liniowej:

```
SELECT
  prd_type_id,
  REGR_AVGX(amount, month) AS avgx,
  REGR_AVGY(amount, month) AS avgy,
  REGR_COUNT(amount, month) AS count,
  REGR_INTERCEPT(amount, month) AS inter,
  REGR_R2(amount, month) AS r2,
  REGR_SLOPE(amount, month) AS slope,
  REGR_SXX(amount, month) AS sxx,
  REGR_SXY(amount, month) AS sxy,
  REGR_SYy(amount, month) AS syy
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id;
```

PRD_TYPE_ID	AVGX	AVGY	COUNT	INTER	R2
SLOPE	SXX	SXY	SYy		
1 -115,05741	6,5 858	12570,5811 -98719,26	72 3031902717	13318,4543 151767392	,003746289
2 -2,997634	6,5 858	2588,62806 -2571,97	72 151767392	2608,11268 126338815	,0000508
3 690,526206	6,5 858	6642,65153 592471,485	72 3238253324	2154,23119 1985337488	,128930297
4 546,199149	6,5 858	5593,76611 468638,87	72 1985337488	2043,47164 126338815	,128930297
5			0		

Użycie funkcji hipotetycznego rankingu i rozkładu

Funkcje hipotetycznego rankingu i rozkładu służą do obliczania pozycji i percentylu, które zająłby nowy wiersz po wstawieniu go do tabeli. Obliczenia hipotetyczne można wykonywać za pomocą następujących funkcji: RANK(), DENSE_RANK(), PERCENT_RANK() i CUME_DIST().

W poniższym zapytaniu użyto funkcji RANK() i PERCENT_RANK() do pobrania pozycji i pozycji procentowej wartości sprzedaży według rodzajów produktów w 2003 roku:

```
SELECT
  prd_type_id, SUM(amount),
  RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
  PERCENT_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	PERCENT_RANK
1	905081,84	1	0
2	186381,22	4	1
3	478270,91	2	,333333333
4	402751,16	3	,666666667

Kolejne zapytanie oblicza hipotetyczną pozycję i pozycję procentową wartości sprzedaży wynoszącej 500 000 zł:

```

SELECT
  RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS rank,
  PERCENT_RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS percent_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;

      RANK PERCENT_RANK
----- -----
      2           ,25

```

Hipotetyczna pozycja i pozycja procentowa wartości sprzedaży równej 500 000 zł wynosi odpowiednio 2 i 0,25.

Na tym zakończymy omówienie funkcji obliczających wartości hipotetyczne.

Użycie klauzuli MODEL

Klauzula MODEL została wprowadzona w Oracle Database 10g i umożliwia wykonywanie obliczeń międzywierszowych. Dzięki niej możliwe jest uzyskanie dostępu do kolumny w wierszu w taki sposób, jakby była to komórka w tablicy. Możemy więc wykonywać obliczenia w sposób przypominający pracę w arkuszu kalkulacyjnym. Na przykład tabela `all_sales` zawiera informacje o sprzedaży w miesiącach 2003 roku. Za pomocą klauzuli MODEL na podstawie wartości sprzedaży w 2003 roku możemy obliczyć wartość sprzedaży w przyszłych miesiącach.

Przykład zastosowania klauzuli MODEL

Sposób działania klauzuli MODEL najłatwiej przedstawić na przykładzie. Poniższe zapytanie pobiera wartości sprzedaży w poszczególnych miesiącach 2003 roku, uzyskane przez pracownika nr 21 dla produktów 1. i 2. rodzaju, a następnie oblicza przewidywaną sprzedaż w styczniu, lutym i marcu 2004 roku na podstawie sprzedaży w 2003 roku:

```

SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[1, 2004] = sales_amount[1, 2003],
  sales_amount[2, 2004] = sales_amount[2, 2003] + sales_amount[3, 2003],
  sales_amount[3, 2004] = ROUND(sales_amount[3, 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;

```

Zapytanie składa się z następujących elementów:

- PARTITION BY (`prd_type_id`) określa, że wyniki są partycjonowane według `prd_type_id`.
- DIMENSION BY (`month, year`) określa, że wymiarami tablicy są `month` i `year`. To oznacza, że dostęp do komórki tablicy uzyskujemy, podając miesiąc i rok.
- MEASURES (`amount sales_amount`) określa, że każda komórka tablicy zawiera kwotę sprzedaży i że nazwa tablicy to `sales_amount`. Aby wydobyć z tablicy `sales_amount` wartość dla stycznia 2003 roku, używamy `sales_amount[1, 2003]`.
- Po klauzuli MEASURES znajdują się trzy wiersze, które obliczają przyszłe kwoty sprzedaży w styczniu, lutym i marcu 2004 roku:

- `sales_samount[1, 2004] = sales_amount[1, 2003]` ustawia jako kwotę sprzedaży w styczniu 2004 roku kwotę sprzedaży w styczniu 2003 roku.
- `sales_amount[2, 2004] = sales_amount[2, 2003] + sales_amount[3, 2003]` ustawia jako kwotę sprzedaży w lutym 2004 roku sumę kwot sprzedaży w lutym i marcu 2003 roku.
- `sales_amount[3, 2004] = ROUND(sales_amount[3, 2003] * 1.25, 2)` ustawia kwotę sprzedaży w marcu 2004 roku jako zaokrąglony iloczyn kwoty sprzedaży w marcu 2003 i liczby 1,25.
- `ORDER BY prd_type_id, year, month` po prostu ustala kolejność wyników zwróconych przez całe zapytanie.

Wynik tego zapytania został przedstawiony poniżej. Należy zauważyć, że zawiera on kwoty sprzedaży typów produktów nr 1 i 2 we wszystkich miesiącach 2003 roku oraz przewidywane kwoty sprzedaży w pierwszych trzech miesiącach 2004 roku (zostały one wyróżnione):

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034,84
1	2003	2	15144,65
1	2003	3	20137,83
1	2003	4	25057,45
1	2003	5	17214,56
1	2003	6	15564,64
1	2003	7	12654,84
1	2003	8	17434,82
1	2003	9	19854,57
1	2003	10	21754,19
1	2003	11	13029,73
1	2003	12	10034,84
1	2004	1	10034,84
1	2004	2	35282,48
1	2004	3	25172,29
2	2003	1	1034,84
2	2003	2	1544,65
2	2003	3	2037,83
2	2003	4	2557,45
2	2003	5	1714,56
2	2003	6	1564,64
2	2003	7	1264,84
2	2003	8	1734,82
2	2003	9	1854,57
2	2003	10	2754,19
2	2003	11	1329,73
2	2003	12	1034,84
2	2004	1	1034,84
2	2004	2	3582,48
2	2004	3	2547,29

Dostęp do komórek za pomocą zapisu pozycyjnego i symbolicznego

W poprzednim przykładzie uzyskaliśmy dostęp do komórki tablicy za pomocą zapisu: `sales_amount[1, 2004]`, w którym 1 oznacza miesiąc, a 2004 — rok. Jest to zapis pozycyjny, ponieważ znaczenie wymiarów jest określone przez ich pozycję: pierwsza zawiera miesiąc, a druga rok.

Do jawnego określenia znaczenia wymiarów możemy użyć zapisu symbolicznego, na przykład `sales_amount[month=1, year=2004]`. Poniższe zapytanie jest zmodyfikowaną wersją wcześniejszego — użyto w nim zapisu symbolicznego:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
```

```

AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[month=1, year=2004] = sales_amount[month=1, year=2003],
    sales_amount[month=2, year=2004] =
        sales_amount[month=2, year=2003] + sales_amount[month=3, year=2003],
    sales_amount[month=3, year=2004] =
        ROUND(sales_amount[month=3, year=2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;

```

Gdy używa się zapisu symbolicznego lub pozycjnego, należy zdawać sobie sprawę, że w różny sposób traktują one wartości NULL w wymiarach. Na przykład `sales_amount[null, 2003]` zwraca kwotę, dla której miesiąc to NULL, a rok to 2003, a `sales_amount[month=null, year=2004]` nie uzyska dostępu do prawidłowej komórki, ponieważ `null=null` zawsze zwraca fałsz.

Uzyskiwanie dostępu do zakresu komórek za pomocą BETWEEN i AND

Za pomocą słów kluczowych BETWEEN i AND możemy uzyskać dostęp do zakresu komórek. Poniższe wyrażenie ustawia kwotę sprzedaży w styczniu 2004 roku jako zaokrągloną średnią kwot sprzedaży między styczniem a marcem 2003 roku:

```

sales_amount[1, 2004] =
    ROUND(AVG(sales_amount)[month BETWEEN 1 AND 3, 2003], 2)

```

Poniższe zapytanie obrazuje usage tego wyrażenia:

```

SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[1, 2004] =
        ROUND(AVG(sales_amount)[month BETWEEN 1 AND 3, 2003], 2)
)
ORDER BY prd_type_id, year, month;

```

Sięganie do wszystkich komórek za pomocą ANY i IS ANY

Za pomocą predykatów ANY i IS ANY możemy uzyskać dostęp do wszystkich komórek tablicy. Predykat ANY jest używany z zapisem pozycyjnym, a IS ANY stosuje się z zapisem symbolicznym. Poniższe wyrażenie ustawia kwotę sprzedaży w styczniu 2004 roku jako zaokrągloną sumę kwot sprzedaży we wszystkich miesiącach i latach:

```

sales_amount[1, 2004] =
    ROUND(SUM(sales_amount)[ANY, year IS ANY], 2)

```

Poniższe zapytanie obrazuje usage tego wyrażenia:

```

SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)

```

```

DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[1, 2004] =
        ROUND(SUM(sales_amount)[ANY, year IS ANY], 2)
)
ORDER BY prd_type_id, year, month;

```

Pobieranie bieżącej wartości wymiaru za pomocą funkcji CURRENTV()

Za pomocą funkcji CURRENTV() możemy pobrać bieżącą wartość wymiaru. Na przykład poniższe wyrażenie ustawia kwotę sprzedaży w pierwszym miesiącu 2004 roku jako iloczyn liczby 1,25 i kwoty sprzedaży w takim samym miesiącu 2003 roku. Funkcję CURRENTV() zastosowano do pobrania bieżącego miesiąca, czyli 1:

```

sales_amount[1, 2004] =
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)

```

Poniższe zapytanie obrazuje użycie tego wyrażenia:

```

SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[1, 2004] =
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;

```

Poniżej zostały przedstawione wyniki tego zapytania — wartości dla 2004 roku zostały pogrubione:

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034,84
1	2003	2	15144,65
1	2003	3	20137,83
1	2003	4	25057,45
1	2003	5	17214,56
1	2003	6	15564,64
1	2003	7	12654,84
1	2003	8	17434,82
1	2003	9	19854,57
1	2003	10	21754,19
1	2003	11	13029,73
1	2003	12	10034,84
1	2004	1	12543,55
2	2003	1	1034,84
2	2003	2	1544,65
2	2003	3	2037,83
2	2003	4	2557,45
2	2003	5	1714,56
2	2003	6	1564,64
2	2003	7	1264,84
2	2003	8	1734,82
2	2003	9	1854,57
2	2003	10	2754,19
2	2003	11	1329,73
2	2003	12	1034,84
2	2004	1	1293,55

Uzyskiwanie dostępu do komórek za pomocą pętli FOR

Dostęp do komórek możemy również uzyskać za pomocą pętli FOR. Na przykład poniższe wyrażenie ustawia kwotę sprzedaży dla pierwszych trzech miesięcy 2004 roku jako iloczyn liczby 1,25 i kwot sprzedaży w odpowiednich miesiącach 2003 roku. Użyto tutaj słowa kluczowego INCREMENT określającego, o ile zostanie zwiększoną wartość month przy każdej iteracji pętli:

```
sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
```

W poniższym zapytaniu zastosowano to wyrażenie:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

Oto wyniki tego zapytania — wartości dla 2004 roku zostały pogrubione:

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034,84
1	2003	2	15144,65
1	2003	3	20137,83
1	2003	4	25057,45
1	2003	5	17214,56
1	2003	6	15564,64
1	2003	7	12654,84
1	2003	8	17434,82
1	2003	9	19854,57
1	2003	10	21754,19
1	2003	11	13029,73
1	2003	12	10034,84
1	2004	1	12543,55
1	2004	2	18930,81
1	2004	3	25172,29
2	2003	1	1034,84
2	2003	2	1544,65
2	2003	3	2037,83
2	2003	4	2557,45
2	2003	5	1714,56
2	2003	6	1564,64
2	2003	7	1264,84
2	2003	8	1734,82
2	2003	9	1854,57
2	2003	10	2754,19
2	2003	11	1329,73
2	2003	12	1034,84
2	2004	1	1293,55
2	2004	2	1930,81
2	2004	3	2547,29

Obsługa wartości NULL i brakujących

W tym podrozdziale opisano, jak za pomocą klauzuli MODEL obsłużyć brakujące wartości oraz wartości NULL.

Użycie IS PRESENT

IS PRESENT zwraca wartość prawda, jeżeli wiersz określany przez odwołanie komórki istniał przed wykonaniem klauzuli MODEL. Na przykład:

```
sales_amount[CURRENTV(), 2003] IS PRESENT
```

zwróci prawdę, jeśli sales_amount[CURRENTV(), 2003] istnieje.

Poniższe wyrażenie ustawia kwotę sprzedaży w pierwszych trzech miesiącach 2004 roku jako iloczyn liczby 1,25 i kwoty sprzedaży w tych samych miesiącach 2003 roku:

```
sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
CASE WHEN sales_amount[CURRENTV(), 2003] IS PRESENT THEN
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
ELSE
    0
END
```

Poniższe zapytanie przedstawia zastosowanie tego wyrażenia:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    CASE WHEN sales_amount[CURRENTV(), 2003] IS PRESENT THEN
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
    ELSE
        0
    END
)
ORDER BY prd_type_id, year, month;
```

Wynik tego zapytania jest taki sam jak w przykładzie z poprzedniego podrozdziału.

Użycie PRESENTV()

PRESENTV(komórka, wyr1, wyr2) zwraca wyrażenie wyr1, jeżeli wiersz określany przez odwołanie komórki istniał przed uruchomieniem klauzuli MODEL. Jeśli wiersz nie istnieje, jest zwracane wyrażenie wyr2. Na przykład:

```
PRESENTV(sales_amount[CURRENTV(), 2003],
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)
```

zwróci zaokrąglone kwoty sprzedaży, jeżeli sales_amount[CURRENTV(), 2003] istnieje; w przeciwnym razie zostanie zwrócone 0.

Poniższe zapytanie stanowi przykład użycia tego wyrażenia:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    PRESENTV(sales_amount[CURRENTV(), 2003],
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)
)
ORDER BY prd_type_id, year, month;
```

Użycie PRESENTNNV()

`PRESENTNNV(komórka, wyr1, wyr2)` zwraca wyrażenie `wyr1`, jeżeli wiersz określany przez odwołanie `komórka` istniał przed wykonaniem klauzuli MODEL, a komórka ma inną wartość niż NULL. Jeżeli wiersz nie istnieje lub komórka ma wartość NULL, jest zwracane wyrażenie `wyr2`. Na przykład:

```
PRESENTNNV(sales_amount[CURRENTV(), 2003],
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)
```

zwróci zaokrąglone kwoty sprzedaży, jeżeli `sales_amount[CURRENTV(), 2003]` istnieje i jest różne od NULL; w przeciwnym razie zostanie zwrócone 0.

Użycie IGNORE NAV i KEEP NAV

Domyślnie klauzula MODEL traktuje komórkę nie posiadającą wartości tak, jakby miała wartość NULL. Komórka z wartością NULL jest traktowana w ten sam sposób. Można zmienić to domyślne zachowanie, używając IGNORE NAV, które zwraca jedną z poniższych wartości:

- 0 dla brakujących lub równych NULL wartości liczbowych,
- pusty napis dla napisów brakujących lub z wartością NULL,
- 00/01/01 dla brakujących lub równych NULL wartości dat,
- NULL dla brakujących lub równych NULL wartości pozostałych typów danych.

Można też jawnie użyć KEEP NAV, które jest wykorzystywane domyślnie. KEEP NAV zwraca NULL dla brakujących wartości liczbowych lub tych równych NULL.

Poniższe zapytanie obrazuje zastosowanie IGNORE NAV:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL IGNORE NAV
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

Modyfikowanie istniejących komórek

Domyślnie, jeżeli komórka, do której następuje odwołanie po lewej stronie wyrażenia, istnieje, jest modyfikowana. Jeżeli nie istnieje, w tablicy tworzony jest nowy wiersz. Za pomocą RULES UPDATE można zmienić to domyślne zachowanie tak, że nowy wiersz nie będzie tworzony.

Poniższe zapytanie jest przykładem zastosowania RULES UPDATE:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount)
RULES UPDATE (
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

Ponieważ komórki dla 2004 roku nie istnieją, a użyto RULES UPDATE, w tablicy nie zostaną utworzone nowe wiersze dla tego roku, dlatego zapytanie nie zwróci dla niego wierszy. Poniżej przedstawiono wyniki tego zapytania — brakuje w nich wierszy dla 2004 roku:

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034,84
1	2003	2	15144,65
1	2003	3	20137,83
1	2003	4	25057,45
1	2003	5	17214,56
1	2003	6	15564,64
1	2003	7	12654,84
1	2003	8	17434,82
1	2003	9	19854,57
1	2003	10	21754,19
1	2003	11	13029,73
1	2003	12	10034,84
2	2003	1	1034,84
2	2003	2	1544,65
2	2003	3	2037,83
2	2003	4	2557,45
2	2003	5	1714,56
2	2003	6	1564,64
2	2003	7	1264,84
2	2003	8	1734,82
2	2003	9	1854,57
2	2003	10	2754,19
2	2003	11	1329,73
2	2003	12	1034,84

Użycie klauzul PIVOT i UNPIVOT

Klauzula PIVOT została wprowadzona w Oracle Database 11g i umożliwia przestawienie w wynikach zapytania wierszy na miejsce kolumn i jednocześnie wykonanie funkcji agregujących na danych. W Oracle Database 11g jest również dostępna klauzula UNPIVOT, która w wynikach zapytania przestawia kolumny w miejsce wierszy.

Klauzule PIVOT i UNPIVOT przydają się, jeżeli chcemy zobaczyć ogólne trendy w dużej liczbie danych, na przykład trendy sprzedaży w czasie.

Prosty przykład klauzuli PIVOT

Sposób użycia klauzuli PIVOT najłatwiej przedstawić na przykładzie. Poniższe zapytanie przedstawia całkowitą kwotę sprzedaży produktów pierwszego, drugiego i trzeciego typu w pierwszych czterech miesiącach 2003 roku. Należy zauważać, że komórki w wynikach zapytania zawierają sumy kwot sprzedaży każdego produktu w każdym miesiącu:

```
SELECT *
FROM (
  SELECT month, prd_type_id, amount
  FROM all_sales
  WHERE year = 2003
  AND prd_type_id IN (1, 2, 3)
)
PIVOT (
  SUM(amount) FOR month IN (1 AS STY, 2 AS LUT, 3 AS MAR, 4 AS KWI)
)
ORDER BY prd_type_id;
```

PRD_TYPE_ID	STY	LUT	MAR	KWI
-------------	-----	-----	-----	-----

1	38909,04	70567,9	91826,98	120344,7
2	14309,04	13367,9	16826,98	15664,7
3	24909,04	15467,9	20626,98	23844,7

Rozpoczynając od pierwszego wiersza wyników, widzimy, że:

- w styczniu sprzedano produktów pierwszego typu za kwotę 38 909,04 zł,
- w lutym sprzedano produktów tego typu za kwotę 70 567,9 zł,
- ... itd. dla całego wiersza.

W drugim wierszu wyników widać, że:

- w styczniu sprzedano produktów drugiego typu za kwotę 14 309,04 zł,
- w lutym sprzedano produktów tego typu za kwotę 13 367,9 zł,
- ... itd. dla pozostałych wyników.

Uwaga

Klauzula PIVOT jest potężnym narzędziem, które umożliwia przedstawienie trendów sprzedaży różnego rodzaju produktów w poszczególnych miesiącach. Na podstawie takich danych w prawdziwym sklepie można podjąć decyzję o zmianie strategii sprzedażowej lub o przygotowaniu nowych kampanii marketingowych.

Poprzednia instrukcja SELECT ma następującą strukturę:

```
SELECT *
FROM (
    zapytanie_wewnętrzne
)
PIVOT (
    funkcja_agregująca FOR kolumna_obrotu IN (lista_wartości)
)
ORDER BY ...;
```

Rozłożymy poprzednie zapytanie na elementy składowe:

- Występuje w nim zapytanie zewnętrzne i wewnętrzne. Wewnętrzne pobiera miesiąc, rodzaj produktu i kwotę z tabeli `a11_sales` i przesyła wyniki do zapytania zewnętrznego.
- `SUM(amount) FOR month IN (1 AS STY, 2 AS LUT, 3 AS MAR, 4 AS KWI)` jest linią w klaузuli PIVOT:
 - Funkcja `SUM()` dodaje kwoty sprzedaży poszczególnych rodzajów produktów w pierwszych czterech miesiącach (wymienionych w części IN). Miesiące w wynikach nie są określone jako 1, 2, 3 i 4, ponieważ część AS zmienia ich nazwy na STY, LUT, MAR i KWI w celu zwiększenia czytelności wyników.
 - Kolumna `month` tabeli `a11_sales` jest używana jako kolumna przestawienia. To oznacza, że w wynikach miesiące pojawiają się jako kolumny. Na skutek tego wiersze są obrócone — czy też *przestawione* — aby były wyświetlane jako kolumny.
 - Na samym końcu przykładu linia `ORDER BY prd_type_id` po prostu porządkuje wyniki według rodzaju produktu.

Przestawianie w oparciu o wiele kolumn

Przestawienie możemy wykonać w oparciu o wiele kolumn, umieszczając ich nazwy w części FOR klauzuli PIVOT. W poniższym przykładzie są przestawiane kolumny `month` i `prd_type_id`, do których występuje odwołanie w części FOR. Należy zauważyc, że lista wartości w części IN klauzuli PIVOT zawiera wartość dla kolumn `month` i `prd_type_id`:

```
SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM a11_sales
    WHERE year = 2003
)
```

```

        AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) FOR (month, prd_type_id) IN (
        (1, 2) AS STY_PRDTYPE2,
        (2, 3) AS LUT_PRDTYPE3,
        (3, 1) AS MAR_PRDTYPE1,
        (4, 2) AS KWI_PRDTYPE2
    )
);
-----
```

STY_PRDTYPE2 LUT_PRDTYPE3 MAR_PRDTYPE1 KWI_PRDTYPE2

	14309,04	15467,9	91826,98	15664,7

Komórki w wynikach zawierają sumę kwot sprzedaży każdego rodzaju produktu w określonym miesiącu (rodzaj produktu i miesiąc są umieszczone na liście wartości w części IN). Jak widać w wynikach zapytania, kwoty sprzedaży były następujące:

- 14 309,04 zł dla produktu drugiego typu w styczniu,
- 15 467,90 zł dla produktu trzeciego typu w lutym,
- 91 826,98 zł dla produktu pierwszego typu w marcu,
- 15 664,70 zł dla produktu drugiego typu w kwietniu.

W części IN możemy umieścić wszystkie interesujące nas wartości. W poniższym przykładzie wartości rodzajów produktów w części IN są wymieszane, aby uzyskać kwoty sprzedaży tych produktów w określonych miesiącach:

```

SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) FOR (month, prd_type_id) IN (
        (1, 1) AS STY_PRDTYPE1,
        (2, 2) AS LUT_PRDTYPE2,
        (3, 3) AS MAR_PRDTYPE3,
        (4, 1) AS KWI_PRDTYPE1
    )
);
-----
```

STY_PRDTYPE1 LUT_PRDTYPE2 MAR_PRDTYPE3 KWI_PRDTYPE1

	38909,04	13367,9	20626,98	120344,7

Uzyskano następujące kwoty sprzedaży:

- 38 909,04 zł dla produktu pierwszego typu w styczniu,
- 13 367,90 zł dla produktu drugiego typu w lutym,
- 20 626,98 zł dla produktu trzeciego typu w marcu,
- 120 344,70 zł dla produktu pierwszego typu w kwietniu.

Użycie kilku funkcji agregujących w przestawieniu

W przestawieniu możemy użyć wielu funkcji agregujących. Na przykład w poniższym zapytaniu użyto funkcji SUM() do obliczenia całkowitej kwoty sprzedaży wybranych rodzajów produktów w styczniu i lutym oraz funkcji AVG() do obliczenia średnich kwot sprzedaży:

```

SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) AS sum_amount,
    AVG(amount) AS avg_amount
    FOR (month) IN (
        1 AS STY, 2 AS LUT
    )
)
ORDER BY prd_type_id;

```

PRD_TYPE_ID	STY_SUM_AMOUNT	STY_AVG_AMOUNT	LUT_SUM_AMOUNT	LUT_AVG_AMOUNT
1	38909,04	6484,84	70567,9	11761,3167
2	14309,04	2384,84	13367,9	2227,98333
3	24909,04	4151,50667	15467,9	2577,98333

Pierwszy wiersz wyników przedstawia kwoty sprzedaży dla produktu pierwszego typu:

- w styczniu całkowita kwota wyniosła 38 909,04 zł przy średniej 6 484,84 zł,
- w lutym całkowita kwota wyniosła 70 567,90 zł przy średniej 11 761,32.

Drugi wiersz wyników przedstawia kwoty sprzedaży dla produktu drugiego typu:

- w styczniu całkowita kwota wyniosła 14 309,04 zł przy średniej 2 384,84 zł,
- w lutym całkowita kwota wyniosła 13 367,90 zł przy średniej 2 227,98.
- ... itd. dla trzeciego wiersza.

Użycie klauzuli UNPIVOT

Klauzula UNPIVOT obraca kolumny w wiersze. Wykonuje ona operację przeciwną do wykonywanej przez klauzulę PIVOT.

W przykładach zawartych w tym podrozdziale korzystano z tabeli pivot_sales_data (tworzonej przez skrypt *store_schema.sql*):

```

CREATE TABLE pivot_sales_data AS
SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) FOR month IN (1 AS STY, 2 AS LUT, 3 AS MAR, 4 AS KWI)
)
ORDER BY prd_type_id;

```

Dane w tabeli pivot_sales_data umieszczane są przez zapytanie zwracające przestawioną wersję danych sprzedaży z tabeli all_sales.

Poniższe zapytanie zwraca zawartość tabeli pivot_sales_data:

```

SELECT *
FROM pivot_sales_data;

```

PRD_TYPE_ID	STY	LUT	MAR	KWI
1	38909,04	70567,9	91826,98	120344,7
2	14309,04	13367,9	16826,98	15664,7
3	24909,04	15467,9	20626,98	23844,7

W kolejnym zapytaniu użyto klauzuli UNPIVOT do pobrania danych sprzedaży w formie nieprzestawionej:

```
SELECT *
FROM pivot_sales_data
UNPIVOT (
    amount FOR month IN (STY, LUT, MAR, KWI)
)
ORDER BY prd_type_id;
```

PRD_TYPE_ID	MON	AMOUNT
1	STY	38909,04
1	LUT	70567,9
1	MAR	91826,98
1	KWI	120344,7
2	STY	14309,04
2	LUT	13367,9
2	KWI	15664,7
2	MAR	16826,98
3	STY	24909,04
3	MAR	20626,98
3	LUT	15467,9
3	KWI	23844,7

W tych wynikach miesięczne wartości sprzedaży są zaprezentowane pionowo. Warto porównać te wyniki z wynikami wcześniejszego zapytania, gdzie wartości sprzedaży były zaprezentowane horyzontalnie.

Zapytania o określona liczbę wierszy

Nowością w Oracle Database 12c jest wbudowane wsparcie dla zapytań o określona liczbę wierszy (ang. *top-N queries*). Zapytanie takie zawiera klauzulę ograniczającą liczbę zwracanych wierszy. Taka klauzula umożliwia ograniczenie ilości pobieranych wierszy przez określenie:

- ilości wierszy do pobrania za pomocą klauzuli **FETCH FIRST**;
- przesunięcia określającego, ile wierszy należy pominąć, zanim rozpocznie się zliczanie wierszy za pomocą klauzuli **OFFSET**;
- części całkowitej ilości wierszy w procentach za pomocą klauzuli **PERCENT**.

Można dodać też kolejne warunki do klauzul ograniczających ilość wierszy:

- **ONLY**, który powoduje zwrócenie dokładnie określonej w klauzuli ilości wierszy;
- **WITH TIES**, który powoduje dołączenie dodatkowych wierszy z taką samą wartością klucza sortowania jak ostatni zwracany wiersz (klucz sortowania to kolumna określona w klauzuli **ORDER BY**).

W kolejnych podrozdziałach pokazane są przykłady.

Użycie klauzuli **FETCH FIRST**

Poniższe zapytanie wykorzystuje **FETCH FIRST** do pobrania pięciu pracowników z najmniejszą wartością identyfikatora `employee_id` z tabeli `more_employees`:

```
SELECT employee_id, first_name, last_name
FROM more_employees
ORDER BY employee_id
FETCH FIRST 5 ROWS ONLY;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	Jan	Kowalski
2	Roman	Joźwierz
3	Fryderyk	Helc
4	Zuzanna	Nowak
5	Robert	Zielony

W tym przykładzie pracownicy uporządkowani są według wartości `employee_id` i zwracane jest pięć wierszy z najniższymi wartościami `employee_id`.

Następne zapytanie pobiera pięciu pracowników z największymi wartościami `employee_id` z tabeli `more_employees`. Dzieje się tak, ponieważ zastosowane zostało porządkowanie malejąco według wartości `employee_id`.

```
SELECT employee_id, first_name, last_name
FROM more_employees
ORDER BY employee_id DESC
FETCH FIRST 5 ROWS ONLY;
```

EMPLOYEE_ID FIRST_NAME LAST_NAME

13	Dorota	Prewał
12	Franciszek	Słaby
11	Kamil	Długi
10	Kazimierz	Czarny
9	Henryk	Borny

Kolejne zapytanie korzysta z `FETCH FIRST`, by pobrać cztery produkty z najniższą ceną z tabeli `products`. Produkty są porządkowane według ceny, a następnie zwracane są cztery wiersze z najniższą ceną.

```
SELECT product_id, name, price
FROM products
ORDER BY price
FETCH FIRST 4 ROWS ONLY;
```

PRODUCT_ID NAME PRICE

9	Muzyka klasyczna	10,99
8	Z innej planety	12,99
7	Space Force 9	13,49
12	Pierwsza linia	13,49

Użycie klauzuli OFFSET

Z pomocą klauzuli `OFFSET` można określić ilość wierszy, jaką należy pominąć przed rozpoczęciem wyświetlanego wyników. Poniższe zapytanie pobiera pracowników z `employee_id` o wartościach od 6 do 10 z tabeli `more_employees`:

```
SELECT employee_id, first_name, last_name
FROM more_employees
ORDER BY employee_id
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;
```

EMPLOYEE_ID FIRST_NAME LAST_NAME

6	Joanna	Brąz
7	Jan	Szary
8	Jadwiga	Niebieska
9	Henryk	Borny
10	Kazimierz	Czarny

Klauzula `OFFSET` w tym przykładzie ma postać:

OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY

Oznacza to, że należy rozpocząć pobieranie wyników po wierszu nr 5 i pobrać pięć kolejnych wierszy po tej pozycji. To powoduje, że zostaną pobrane wiersze z wartościami `employee_id` od 6 do 10.

Następne zapytanie korzysta z klauzuli `offset` do pobrania produktów z tabeli `products`. Zapytanie szereguje produkty według ceny, rozpoczyna po wierszu nr 5 i pobiera kolejne cztery wiersze po tej pozycji.

```
SELECT product_id, name, price
FROM products
```

```
ORDER BY price
OFFSET 5 ROWS FETCH NEXT 4 ROWS ONLY;
```

PRODUCT_ID	NAME	PRICE
6	2412: Powrót	14,95
11	Twórczy wrzask	14,99
10	Pop 3	15,99
1	Nauka współczesna	19,95

Użycie klauzuli PERCENT

Klauzuli PERCENT można użyć do wskazania, jaki procent wszystkich wierszy wyniku zapytania należy zwrócić. Poniższe zapytanie pobiera 20 procent produktów z najwyższą ceną z tabeli products:

```
SELECT product_id, name, price
FROM products
ORDER BY price DESC
FETCH FIRST 20 PERCENT ROWS ONLY;
```

PRODUCT_ID	NAME	PRICE
5	Z Files	49,99
2	Chemia	30
3	Supernowa	25,99

Kolejne zapytanie pobiera 10 procent pracowników z najniższym wynagrodzeniem z tabeli more_employees:

```
SELECT employee_id, first_name, last_name, salary
FROM more_employees
ORDER BY salary
FETCH FIRST 10 PERCENT ROWS ONLY;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
8	Jadwiga	Niebieska	29000
7	Jan	Szary	30000

Użycie klauzuli WITH TIES

Klauzuli WITH TIES można użyć, by dołączyć dodatkowe wiersze z tą samą wartością klucza sortowania jak ostatni z pobieranych wierszy. Klucz sortowania to kolumna wskazana w klauzuli ORDER BY.

Poniższe zapytanie pobiera 10 procent pracowników z najniższym wynagrodzeniem z tabeli more_employees za pomocą WITH TIES. Kluczem sortowania jest kolumna salary:

```
SELECT employee_id, first_name, last_name, salary
FROM more_employees
ORDER BY salary
FETCH FIRST 10 PERCENT ROWS WITH TIES;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
8	Jadwiga	Niebieska	29000
7	Jan	Szary	30000
9	Henryk	Borný	30000

Porównaj te wyniki z wynikami z poprzedniego przykładu. W tych wynikach pojawił się pracownik Henryk Borný, który ma takie samo wynagrodzenie jak pracownik z ostatniego wiersza zwróconego w poprzednim przykładzie.

Odnajdywanie wzorców w danych

Nową funkcjonalnością Oracle Database 12c jest natywne wsparcie poszukiwania wzorca w danych za pomocą klauzuli MATCH_RECOGNIZE. Odnajdywanie wzorców w danych jest przydatne w wielu sytuacjach. Na przykład odnajdywanie wzorców w danych jest użyteczne przy poszukiwaniu trendów w sprzedaży, wykrywaniu oszustw w transakcjach dokonywanych za pomocą kart kredytowych czy odkrywaniu włamań do sieci.

Przykłady w tym podrozdziale pokazują, jak odnajdywać wzorce formacji typu V i typu W w wielkości sprzedaży produktu w ciągu kilku dni. Te wzorce mogą być użyte na przykład do lepszego wybrania czasu kampanii marketingowych. Przykłady te korzystają z tabel all_sales2 i all_sales3 tworzonych przez skrypt store_schema.sql za pomocą następujących instrukcji:

```
CREATE TABLE all_sales2 (
    product_id INTEGER REFERENCES products(product_id),
    total_amount NUMBER(8, 2),
    sale_date DATE
);

CREATE TABLE all_sales3 (
    product_id INTEGER REFERENCES products(product_id),
    total_amount NUMBER(8, 2),
    sale_date DATE
);
```

Obie tabele zawierają takie same kolumny:

- **product_id** to identyfikator sprzedanego produktu,
- **total_amount** to całkowita cena produktu sprzedanego w ciągu dnia,
- **sale_date** to data sprzedaży.

W kolejnych podrozdziałach dowiesz się, jak odnajdować w danych wzorce formacji typu V i typu W.

Odnajdywanie wzorców formacji typu V w danych z tabeli all_sales2

W tym podrozdziale dowiesz się, jak odnaleźć formacje typu V (ang. *V-shaped data patterns*) w tabeli all_sales2. Dla uproszczenia w tabeli zapisane są dane o wielkości sprzedaży jednego produktu w ciągu dziesięciu dni.

Poniższe zapytanie pobiera wiersze tabeli all_sales2. Wartość **total_amount** osiąga najwyższą wartość 3 czerwca, następnie spada do 7 czerwca, potem rośnie do 9 czerwca. Te daty to odpowiednio: początkowy, najwyższy i końcowy punkt formacji typu V w danych sprzedażowych (odpowiednie wiersze pogrubilem w wynikach zapytania).

```
SELECT *
FROM all_sales2;

PRODUCT_ID TOTAL_AMOUNT SALE_DATE
-----
1          1000 11/06/01
1          1100 11/06/02
1          1200 11/06/03
1          1100 11/06/04
1          1000 11/06/05
1          900 11/06/06
1          800 11/06/07
1          900 11/06/08
1          1000 11/06/09
1          900 11/06/10
```

Poniższe zapytanie odnajduje wzorzec formacji typu V i zwraca daty wystąpienia początkowego (`strt`), najniższego (down) i końcowego (end) punktu:

```

SELECT *
FROM all_sales2
MATCH_RECOGNIZE (
    PARTITION BY product_id
    ORDER BY sale_date
    MEASURES
        strt.sale_date AS start_v_date,
        down.sale_date AS low_v_date,
        up.sale_date AS end_v_date
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO LAST up
    PATTERN (strt down+ up+)
    DEFINE
        down AS down.total_amount < PREV(down.total_amount),
        up AS up.total_amount > PREV(up.total_amount)
) my_pattern
ORDER BY my_pattern.product_id, my_pattern.start_v_date;

PRODUCT_ID START_V_DATE LOW_V_DATE END_V_DATE
----- -----
1 11/06/03 11/06/07 11/06/09

```

Przeanalizujmy każdą część klauzuli `MATCH_RECOGNIZE` w tym przykładzie:

- `PARTITION BY product_id` grupuje wiersze pobrane z tabeli `all_sales2` według wartości w kolumnie `product_id`. Każda grupa wierszy zawiera tę samą wartość `product_id`. (Tabela `all_sales2` opisuje sprzedaż tylko jednego produktu, ale w prawdziwych danych mogą znajdować się dane o sprzedaży tysięcy różnych produktów).
- `ORDER BY sale_date` sortuje wiersze w każdej grupie według daty sprzedaży zapisanej w kolumnie `sale_date`.
- `MEASURES` określa następujące elementy zwarcane wśród wyników:
 - `strt.sale_date AS start_v_date`, która jest datą początkową formacji typu V;
 - `down.sale_date AS low_v_date`, która jest datą najniższego punktu tej formacji;
 - `up.sale_date AS end_v_date`, która jest datą końcową formacji.
- `ONE ROW PER MATCH` generuje jeden wiersz w zestawie wyników dla każdej odnalezionej w danych formacji typu V. W tym przykładzie znajduje się tylko jedna taka formacja i dlatego zapytanie tworzy tylko jeden wiersz wyników. Gdyby było więcej tego typu formacji, w wynikach pojawiłby się oddzielny wiersz dla każdej z nich.
- `AFTER MATCH SKIP TO LAST up` powoduje, że wyszukiwanie jest kontynuowane od wiersza zawierającego koniec ostatnio znalezionej wzorca. `up` to zmienna wzorca opisana w klauzuli `define`.
- `PATTERN (strt down+ up+)` wskazuje, że wzorcem do wyszukania w tym przypadku jest formacja typu V. Aby wyobrazić sobie ten wzorzec, należy przeanalizować zapisane w klauzuli `PATTERN` zmienne wzorca: `strt`, `down`, `up` (start, dół, góra). Kolejność tych zmiennych wskazuje kolejność wyszukiwania zmiennych. Symbol plusa (+) umieszczony po zmiennych `down` i `up` wskazuje, że przynajmniej jeden wiersz musi być odnaleziony dla każdej zmiennej. W tym przykładzie klauzula `PATTERN` wyszukuje formacje typu V: początkowy wiersz `strt`, po którym następują najniższe wartości aż do najniższej (`down`), a następnie wyższe aż do najwyższej wartości (`up`).
- `DEFINE` opisuje następujące zmienne wzorca:
 - `down AS down.total_amount < PREV(down.total_amount)`, która wykorzystuje funkcję `PREV()` do porównania wartości w kolumnie `total_amount` bieżącego wiersza z wartością w kolumnie `total_amount` poprzedniego wiersza; wartość `down` jest zapisywana, gdy wartość tej kolumny

w bieżącym wierszu jest mniejsza niż w poprzednim wierszu; zmienna ta opisuje najwyższy punkt formacji typu V.

- up AS up.total_amount > PREV(up.total_amount) opisuje zakończenie formacji typu V.

Poniższe zapytanie pokazuje dodatkowe opcje MATCH_RECOGNIZE:

```

SELECT
    sale_date, start_v_date, low_v_date, end_v_date,
    match_variable, up_days, down_days, count_days,
    sales_difference, total_amount
FROM all_sales2
MATCH_RECOGNIZE (
    PARTITION BY product_id
    ORDER BY sale_date
    MEASURES
        strt.sale_date AS start_v_date,
        FINAL LAST(down.sale_date) AS low_v_date,
        FINAL LAST(up.sale_date) AS end_v_date,
        CLASSIFIER() AS match_variable,
        FINAL COUNT(up.sale_date) AS up_days,
        FINAL COUNT(down.sale_date) AS down_days,
        RUNNING COUNT(sale_date) AS count_days,
        total_amount - strt.total_amount AS sales_difference
    ALL ROWS PER MATCH
    AFTER MATCH SKIP TO LAST up
    PATTERN (strt down+ up+)
    DEFINE
        down AS down.total_amount < PREV(down.total_amount),
        up AS up.total_amount > PREV(up.total_amount)
    ) my_pattern
ORDER BY my_pattern.product_id, my_pattern.start_v_date;

SALE_DATE START_V_DATE LOW_V_DATE END_V_DATE MATCH_VARIABLE
-----+
UP_DAYS DOWN_DAYS COUNT_DAYS SALES_DIFFERENCE TOTAL_AMOUNT
-----+
11/06/03 11/06/03 11/06/07 11/06/09 STRT
      2          4          1            0         1200
11/06/04 11/06/03 11/06/07 11/06/09 DOWN
      2          4          2           -100       1100
11/06/05 11/06/03 11/06/07 11/06/09 DOWN
      2          4          3           -200       1000
11/06/09 11/06/03 11/06/07 11/06/09 UP
      2          4          7           -200       1000
11/06/07 11/06/03 11/06/07 11/06/09 DOWN
      2          4          5           -400       800
11/06/08 11/06/03 11/06/07 11/06/09 UP
      2          4          6           -300       900
11/06/06 11/06/03 11/06/07 11/06/09 DOWN
      2          4          4           -300       900

```

Przeanalizujmy najważniejsze elementy klauzuli MATCH_RECOGNIZE z tego przykładu:

- MEASURES określa zmienne zwarcane w wynikach:
 - strt.sale_date AS start_v_date, która opisuje datę początkową formacji typu V.
 - FINAL LAST(down.sale_date) AS low_v_date, która opisuje datę w najwyższym punkcie formacji typu V. Funkcja LAST() zwraca ostatnią wartość. Połączenie FINAL() i LAST() dla zmiennej

`low_v_date` powoduje, że wszystkie wiersze opisujące formacje mają taką samą datę najniższego punktu.

- `FINAL LAST(up.sale_date) AS end_v_date` opisuje końcową datę formacji typu V. Połączenie `FINAL` i `LAST()` dla `end_v_date` powoduje, że wszystkie wiersze opisujące formacje mają taką samą datę punktu końcowego.
- `CLASSIFIER() AS match_variable`, która opisuje zmienną dopasowaną w każdym wierszu. W przykładowym zapytaniu zmienne dopasowane to `strt`, `down` i `up`. W wynikach zmienna dopasowana jest wyświetlana wielkimi literami.
- `FINAL COUNT(up.sale_date) AS up_days` opisuje ilość dni dopasowanych do wzorca opisującego wzrost wartości w każdej z odnalezionych formacji typu V.
- `FINAL COUNT(down.sale_date) AS down_days` opisuje ilość dni dopasowanych do wzorca opisującego spadek wartości w każdej z odnalezionych formacji typu V.
- `RUNNING COUNT(sale_date) as count_days`, która zawiera bieżącą sumę dni każdego z okresów zawierających formację typu V.
- `total_amount - strt.total_amount AS sales_difference` opisuje różnicę pomiędzy wartością `total_amount` danego dnia a wartością `total_amount` pierwszego dnia formacji typu V.
- `ALL ROWS PER MATCH` generuje jeden wiersz wyników dla każdego wiersza formacji typu V. W przykładzie znajduje się siedem wierszy w formacji typu V, co oznacza, że zapytanie wygeneruje siedem wierszy wyników.

Odnajdywanie formacji typu W w danych z tabeli all_sales3

W tym podrozdziale dowiesz się, jak odnaleźć formacje typu W w tabeli `all_sales3`. Dla uproszczenia w tabeli znajdują się informacje o sprzedaży tylko jednego produktu w ciągu dziesięciu dni.

Poniższe zapytanie pobiera wiersze tabeli `all_sales3`. Najniższe i najwyższe punkty tworzące formację typu W w kolumnie `total_amount` są wyróżnione w wynikach zapytania. Formacja typu W rozpoczyna się 2 czerwca i kończy 9 czerwca.

```
SELECT *
FROM all_sales3;

PRODUCT_ID TOTAL_AMOUNT SALE_DATE
-----
1          1000 11/06/01
1          1100 11/06/02
1          900 11/06/03
1          800 11/06/04
1          900 11/06/05
1          1000 11/06/06
1          900 11/06/07
1          800 11/06/08
1          1000 11/06/09
1          900 11/06/10
```

Poniższe zapytanie odnajduje formacje typu W i zwraca początkową i końcową datę formacji:

```
SELECT *
FROM all_sales3
MATCH_RECOGNIZE (
  PARTITION BY product_id
  ORDER BY sale_date
  MEASURES
    strt.sale_date AS start_w_date,
    up.sale_date AS end_w_date
  ONE ROW PER MATCH
  AFTER MATCH SKIP TO LAST up
  PATTERN (strt down+ up+ down+ up+)
```

```

DEFINE
    down AS down.total_amount < PREV(down.total_amount),
    up AS up.total_amount > PREV(up.total_amount)
) my_pattern
ORDER BY my_pattern.product_id, my_pattern.start_w_date;

PRODUCT_ID START_W_ END_W_DA
-----
1 11/06/02 11/06/09

```

Przeanalizujmy odpowiednie części klauzuli MATCH_RECOGNIZE w poniższym przykładzie:

- MEASURES określa następujące elementy do zwracenia w wynikach zapytania:
 - strt.sale_date AS start_w_date opisuje początkową datę formacji,
 - up.sale_date AS end_w_date opisuje końcową datę formacji.
- PATTERN (strt down+ up+ down+ up+) określa wzorzec do wyszukiwania formacji typu W: początkowy wiersz strt, a po nim seria danych z wartościami malejącymi (down), rosnącymi (up), malejącymi (down) i znowu rosnącymi (up).

Odnajdywanie formacji typu V w tabeli all_sales3

W tym podrozdziale zobaczysz, jak można odnaleźć wzorce formacji typu V w tabeli all_sales3 i interpretację tych wyników. W poprzednim podrozdziale pokazane było, że tabela all_sales3 zawiera wzorzec formacji typu W. Można przyjąć, że formacja typu W to dwie formacje typu V.

Poniższe zapytanie zwraca daty początkowego, najniższego i końcowego punktu formacji typu V w tabeli all_sales3. Zapytanie wykorzystuje ONE ROW PER MATCH, by wygenerować jeden wiersz wyników dla każdego wzorca typu V odnalezionego w danych:

```

SELECT *
FROM all_sales3
MATCH_RECOGNIZE (
    PARTITION BY product_id
    ORDER BY sale_date
    MEASURES
        strt.sale_date AS start_v_date,
        down.sale_date AS low_v_date,
        up.sale_date AS end_v_date
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO LAST up
    PATTERN (strt down+ up+)
    DEFINE
        down AS down.total_amount < PREV(down.total_amount),
        up AS up.total_amount > PREV(up.total_amount)
    ) my_pattern
ORDER BY my_pattern.product_id, my_pattern.start_v_date;

PRODUCT_ID START_V_ LOW_V_DA END_V_DA
-----
1 11/06/02 11/06/04 11/06/06
1 11/06/06 11/06/08 11/06/09

```

Wyniki zawierają dwa wiersze, po jednym dla każdego wzorca formacji typu V znalezionejego w danych.

Poniższe zapytanie wykorzystuje klauzulę ALL ROWS PER MATCH, która generuje jeden wiersz wyników dla każdego wiersza należącego do formacji typu V:

```

SELECT *
FROM all_sales3
MATCH_RECOGNIZE (
    PARTITION BY product_id
    ORDER BY sale_date
    MEASURES
        strt.sale_date AS start_v_date,

```

```

down.sale_date AS low_v_date,
up.sale_date AS end_v_date
ALL ROWS PER MATCH
AFTER MATCH SKIP TO LAST up
PATTERN (strt down+ up+)
DEFINE
  down AS down.total_amount < PREV(down.total_amount),
  up AS up.total_amount > PREV(up.total_amount)
) my_pattern
ORDER BY my_pattern.product_id, my_pattern.start_v_date;

```

PRODUCT_ID	SALE_DATE	START_V_DATE	LOW_V_DATE	END_V_DATE	TOTAL_AMOUNT
1	11/06/02	11/06/02			1100
1	11/06/03	11/06/02	11/06/03		900
1	11/06/04	11/06/02	11/06/04		800
1	11/06/05	11/06/02	11/06/04	11/06/05	900
1	11/06/06	11/06/02	11/06/04	11/06/06	1000
1	11/06/06	11/06/06			1000
1	11/06/07	11/06/06	11/06/07		900
1	11/06/08	11/06/06	11/06/08		800
1	11/06/09	11/06/06	11/06/08	11/06/09	1000

W wynikach znajduje się w sumie dziewięć wierszy. Na początku pięć wierszy opisujących pierwszą formację typu V. Na końcu cztery wiersze opisujące drugą formację typu V.

W tym podrozdziale pokazany został użyteczny zestaw opcji służących do odnajdywania wzorców w danych. Kompletny zestaw opcji jest zbyt duży, by opisać go w tej książce. Pełny zestaw informacji można znaleźć w dokumencie *Oracle Database Data Warehousing Optimization Guide 12c Release* opublikowanym przez Oracle Corporation.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- funkcje analityczne umożliwiają wykonywanie złożonych obliczeń,
- klauzula MODEL wykonuje obliczenia międzywierszowe i umożliwia traktowanie danych tabeli jako tablicy,
- klauzule PIVOT i UNPIVOT są przydatne do analizy ogólnych trendów w dużej ilości danych,
- można wykonywać zapytania zwracające ograniczoną ilość wierszy wyników,
- klauzula MATCH_RECOGNIZE jest wykorzystywana do odnajdywania wzorców w danych.

W kolejnym rozdziale zajmiemy się zmienianiem zawartości tabeli.

ROZDZIAŁ

9

Zmienianie zawartości tabeli

Z tego rozdziału dowiesz się, jak wykonywać następujące operacje:

- dodawać, modyfikować i usuwać wiersze za pomocą instrukcji **INSERT**, **UPDATE** i **DELETE**,
- używać transakcji, które mogą składać się z wielu instrukcji SQL,
- utrzymywać wyniki transakcji za pomocą instrukcji **COMMIT** lub też cofać wyniki za pomocą instrukcji **ROLLBACK**,
- wykorzystywać zapytania retrospektywne (ang. *flashback query*) do przeglądania wierszy w pierwotnej postaci, przed dokonaniem zmian.



Przed wykonaniem przykładów z tego rozdziału należy wykonać ponownie skrypt *store_schema.sql* opisany w rozdziale 1., w podrozdziale „Tworzenie schematu bazy danych sklepu”. Dzięki temu wyniki wykonywanych w bazie zapytań będą zgodne z zaprezentowanymi w treści tego rozdziału.

Wstawianie wierszy za pomocą instrukcji **INSERT**

Instrukcja **INSERT** służy do wstawiania wierszy do tabeli. Możemy w niej określić następujące informacje:

- tabelę, do której będzie wstawiany wiersz,
- listę kolumn, dla których chcemy określić wartości,
- listę wartości do zapisania w określonych kolumnach.

Wstawiając wiersz, zwykle przesyłamy wartość dla klucza głównego i wszystkich pozostałych kolumn zdefiniowanych jako **NOT NULL**. Nie musimy określać wartości dla kolumn **NULL**, domyślnie zostanie w nich jednak zapisana wartość **NULL**.

Możemy sprawdzić, które kolumny są zdefiniowane jako **NOT NULL**, korzystając z polecenia **DESCRIBE** SQL*Plus. W poniższym przykładzie wyświetlono opis tabeli **customers**:

DESCRIBE customers

Name	NULL?	Type
CUSTOMER_ID	NOT NULL	NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)

Kolumny `customer_id`, `first_name` i `last_name` są zdefiniowane jako `NOT NULL`, co oznacza, że konieczne jest podanie dla nich wartości. Kolumny `dob` i `phone` nie wymagają przesyłania wartości. Jeżeli te wartości zostaną pominięte przy wstawianiu wiersza, zapisane zostanie `NULL`.

Poniższa instrukcja `INSERT` wstawia wiersz do tabeli `customers`. Należy zauważyć, że kolejność wartości w klauzuli `VALUES` jest zgodna z tą, w jakiej kolumny są wymienione na liście kolumn.

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    6, 'Fryderyk', 'Brąz', '70/01/01', '800-555-1215'
);
```

1 row created.

Został wyświetlony komunikat o utworzeniu jednego wiersza. Możemy to sprawdzić, uruchamiając instrukcję `SELECT`:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
6	Fryderyk	Brąz	70/01/01	800-555-1215

Nowy wiersz jest wyświetlany w wynikach zapytania.

Pomijanie listy kolumn

Listę kolumn można pominąć, jeżeli zostaną przesłane wartości dla wszystkich kolumn, tak jak w poniższym przykładzie:

```
INSERT INTO customers
VALUES (7, 'Joanna', 'Zielona', '70/01/01', '800-555-1216');
```

Jeżeli lista kolumn zostanie pominięta, kolejność przesyłanych wartości musi być zgodna z kolejnością kolumn wypisywanych przez polecenie `DESCRIBE`.

Określanie wartości `NULL` dla kolumny

Za pomocą słowa kluczowego `NULL` możemy przypisać kolumnie wartość `NULL`. Na przykład poniższa instrukcja `INSERT` zapisuje wartość `NULL` w kolumnach `dob` i `phone`:

```
INSERT INTO customers
VALUES (8, 'Zofia', 'Biel', NULL, NULL);
```

Po wyświetleniu tego wiersza za pomocą zapytania nie będą widoczne wartości kolumn `dob` i `phone`, ponieważ są to wartości `NULL`:

```
SELECT *
FROM customers
WHERE customer_id = 8;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
8	Zofia	Biel		

Kolumny `dob` i `phone` są puste.

Umieszczanie pojedynczych i podwójnych cudzysłów w wartościach kolumn

W wartości kolumny możemy umieścić pojedyncze i podwójne cudzysłowy. Na przykład poniższa instrukcja INSERT określa nazwisko nowego klienta jako O'Malley. Użyto w niej dwóch pojedynczych cudzysłów za literą O w nazwisku:

```
INSERT INTO customers
VALUES (9, 'Kyle', 'O''Malley', NULL, NULL);
```

Kolejny przykład określa nazwę produktu jako "Wielki" Gatsby:

```
INSERT INTO products (
    product_id, product_type_id, name, description, price
) VALUES (
    13, 1, '"Wielki" Gatsby', NULL, 12.99
);
```

Kopiowanie wierszy z jednej tabeli do innej

Kopiowanie wierszy z jednej tabeli do innej jest możliwe. W tym celu w instrukcji INSERT należy zastosować zapytanie w miejscu wartości kolumn. Liczba i typy kolumn w tabeli źródłowej oraz docelowej muszą być zgodne. W poniższym przykładzie użyto instrukcji SELECT do pobrania kolumn first_name i last_name pierwszego klienta i przesłania ich do instrukcji INSERT:

```
INSERT INTO customers (customer_id, first_name, last_name)
SELECT 10, first_name, last_name
FROM customers
WHERE customer_id = 1;
```

Należy zauważyć, że customer_id nowego wiersza został ustawiony na 10.



Instrukcja MERGE umożliwia dołączanie wierszy z jednej tabeli do innej. Jest ona znacznie wygodniejsza od łączenia instrukcji INSERT i SELECT w celu skopiowania wierszy z jednej tabeli do innej. Więcej informacji na ten temat znajduje się w podrozdziale „Scalanie wierszy za pomocą instrukcji MERGE”.

Modyfikowanie wierszy za pomocą instrukcji UPDATE

Instrukcja UPDATE służy do modyfikowania wierszy tabeli. Zwykle określa się w niej następujące informacje:

- nazwę tabeli,
- klauzulę WHERE, określającą wiersze, które mają zostać zmienione,
- listę nazw kolumn wraz z nowymi wartościami, określana za pomocą klauzuli SET.

Za pomocą jednej instrukcji UPDATE możemy zmienić jeden lub kilka wierszy. Jeżeli określmy kilka wierszy, ta sama zmiana zostanie wprowadzona we wszystkich tych wierszach. Na przykład poniższa instrukcja UPDATE wpisuje do kolumny last_name słowo Pomarańcza w wierszu, w którym customer_id ma wartość 2:

```
UPDATE customers
SET last_name = 'Pomarańcza'
WHERE customer_id = 2;
```

1 row updated.

SQL*Plus potwierdza, że jeden wiersz został zmodyfikowany. Poniższe zapytanie potwierdza dokonanie modyfikacji:

```
SELECT *
FROM customers
WHERE customer_id = 2;

CUSTOMER_ID FIRST_NAME LAST_NAME      DOB      PHONE
----- ---------
2 Lidia      Pomarańcza 68/02/05 800-555-1212
```

Za pomocą jednej instrukcji UPDATE możemy zmienić wiele wierszy i wiele kolumn. Na przykład późniejsza instrukcja UPDATE podnosi o 20 procent cenę wszystkich produktów, których aktualna cena wynosi 20 zł lub więcej. Ta instrukcja zmienia również wielkość liter w nazwach tych produktów na małe:

```
UPDATE products
SET
  price = price * 1.20,
  name = LOWER(name)
WHERE
  price >= 20;

3 rows updated.
```

Instrukcja zmodyfikowała trzy wiersze. Poniższe zapytanie potwierdza dokonanie zmian:

```
SELECT product_id, name, price
FROM products
WHERE price >= (20 * 1.20);

PRODUCT_ID NAME          PRICE
-----  -----
2 chemia           36
3 supernowa       31,19
5 z_files          59,99
```

W instrukcji UPDATE można również użyć podzapytania. Zostało to opisane w podrozdziale „Pisanie instrukcji UPDATE zawierającej podzapytanie” rozdziału 6.

Klauzula RETURNING

W Oracle Database 10g i nowszych możemy zastosować klauzulę RETURNING do zwracenia wartości z funkcji agregującej, takiej jak AVG(). Funkcje agregujące zostały opisane w rozdziale 4.

Kolejny przykład wykonuje następujące zadania:

- deklaruje zmienną o nazwie average_product_price,
- zmniejsza wartość kolumny price w wierszach tabeli products i za pomocą klauzuli RETURNING zapisuje średnią cenę w zmiennej average_product_price,
- wypisuje wartość zmiennej average_product_price.

```
VARIABLE average_product_price NUMBER
```

```
UPDATE products
SET price = price * 0.75
RETURNING AVG(price) INTO :average_product_price;
12 rows updated.

PRINT average_product_price
AVERAGE_PRODUCT_PRICE
-----
16,1216667
```

Usuwanie wierszy za pomocą instrukcji DELETE

Instrukcja DELETE służy do usuwania wierszy z tabeli. Zwykle należy określić klauzulę WHERE, która ograniczy wiersze przeznaczone do usunięcia. W przeciwnym razie zostaną usunięte *wszystkie* wiersze.

Poniższa instrukcja usuwa z tabeli wiersz, w którym `customer_id` ma wartość 10:

```
DELETE FROM customers
WHERE customer_id = 10;
```

1 row deleted.

SQL*Plus potwierdza, że jeden wiersz został usunięty.

W instrukcji `DELETE` można również zastosować podzapytanie, co zostało opisane w podrozdziale „Pisanie instrukcji `DELETE` zawierającej podzapytanie”, w rozdziale 6.



Jeżeli wykonałeś którykolwiek z powyższych instrukcji uruchom ponownie skrypt `store_schema.sql` w celu ponownego utworzenia wszystkich elementów. Dzięki temu rezultaty otrzymywane w kolejnych zapytaniach będą takie same jak zaprezentowane w książce.

Integralność bazy danych

Przy uruchamianiu instrukcji DML (na przykład `INSERT`, `UPDATE` lub `DELETE`) baza danych zapewnia, że wiersze tabel zachowają spójność. To oznacza, że zmiany wprowadzane w wierszach nie wpływają na zależności klucza głównego i obcych tabel.

Wymuszanie więzów klucza głównego

Przyjrzyjmy się kilku przykładom wymuszania więzów klucza głównego. Kluczem głównym tabeli `customers` jest kolumna `customer_id`, co oznacza, że wszystkie wartości w tej kolumnie muszą być unikatowe. Jeżeli spróbujemy wstawić wiersz z duplikatem wartości klucza głównego, zostanie zwrócony błąd `ORA-00001`, tak jak w poniższym przykładzie:

```
SQL> INSERT INTO customers (
  2   customer_id, first_name, last_name, dob, phone
  3 ) VALUES (
  4   1, 'Jan', 'Cenny', '60/01/01', '800-555-1211'
  5 );
INSERT INTO customers (
*
BŁĄD w linii 1:
ORA-00001: naruszonono więzy unikatowe (STORE.CUSTOMERS_PK)
```

Jeżeli spróbujemy zmienić wartość klucza głównego na taką, która już istnieje w tabeli, zostanie zwrócony ten sam błąd:

```
SQL> UPDATE customers
  2 SET customer_id = 1
  3 WHERE customer_id = 2;
UPDATE customers
*
BŁĄD w linii 1:
ORA-00001: naruszonono więzy unikatowe (STORE.CUSTOMERS_PK)
```

Wymuszanie więzów kluczy obcych

Relacja klucza obcego to taka, w której w jednej tabeli znajduje się odwołanie do kolumny z innej tabeli. Na przykład kolumna `product_type_id` w tabeli `products` odwołuje się do kolumny `product_type_id` w tabeli `product_types`. Tabelę `product_types` nazywamy tabelą **nadrzędną**, a tabelę `products` — **podrzędną**, co obrazuje zależność kolumny `product_type_id` tabeli `products` od kolumny `product_type_id` tabeli `product_types`.

Jeżeli spróbujemy wstawić do tabeli `products` wiersz zawierający nieistniejący `product_type_id`, baza danych zwróci błąd `ORA-02291`. Oznacza on, że system nie potrafił odnaleźć zgodnej wartości klucza nadrzędnego (kluczem nadrzędnym jest kolumna `product_type_id` tabeli `product_types`). W poniższym

przykładzie zwracany jest błąd, ponieważ w tabeli `product_types` nie istnieje wiersz, w którym `product_type_id` ma wartość 6:

```
SQL> INSERT INTO products (
 2   product_id, product_type_id, name, description, price
 3 ) VALUES (
 4   13, 6, 'Test', 'Test', NULL
 5 );
INSERT INTO products (
*
BŁĄD w linii 1:
```

ORA-02291: naruszono więzy spójności (STORE.PRODUCTS_FK_PRODUCT_TYPES) – nie znaleziono klucza nadzawanego

Jeżeli spróbujemy zmodyfikować wartość `product_type_id` w wierszu tabeli `products`, podając nieistniejącą wartość klucza nadzawanego, baza danych zwróci ten sam błąd:

```
SQL> UPDATE products
 2   SET product_type_id = 6
 3 WHERE product_id = 1;
UPDATE products
*
BŁĄD w linii 1:
```

ORA-02291: naruszono więzy spójności (STORE.PRODUCTS_FK_PRODUCT_TYPES) – nie znaleziono klucza nadzawanego

Jeśli spróbujemy ponadto usunąć w tabeli nadzawanej wiersz, który posiada wiersze podrzędne, baza danych zwróci błąd ORA-02292. Na przykład jeżeli spróbujemy usunąć z tabeli `product_types` wiersz, w którym `product_type_id` ma wartość 1, baza danych zwróci ten błąd, ponieważ tabela `products` zawiera wiersze, w których `product_type_id` ma wartość 1:

```
SQL> DELETE FROM product_types
 2 WHERE product_type_id = 1;
DELETE FROM product_types
*
BŁĄD w linii 1:
ORA-02292: naruszono więzy spójności (STORE.PRODUCTS_FK_PRODUCT_TYPES) - znaleziono rekord podrzędny
```

Gdyby system bazy danych zezwalał na tego typu usunięcie, wiersze podrzędne nie wskazywałyby na prawidłowe wartości w tabeli nadzawanej.

Użycie wartości domyślnych

Możesz definiować wartości domyślne dla kolumn. Na przykład poniższa instrukcja tworzy tabelę o nazwie `order_status`. Domyślną wartością kolumny `status` jest 'Zamówienie złożone', a wartością domyślną kolumny `last_modified` jest data i godzina zwracane przez `SYSDATE`:

```
CREATE TABLE order_status (
  order_status_id INTEGER
    CONSTRAINT default_example_pk PRIMARY KEY,
  status VARCHAR2(20) DEFAULT 'Zamówienie złożone' NOT NULL,
  last_modified DATE DEFAULT SYSDATE
);
```



Tabela `order_status` jest tworzona przez skrypt `store_schema.sql`. To oznacza, że nie jest konieczne samodzielne wpisywanie powyższej instrukcji `CREATE TABLE`. Nie jest także konieczne wpisywanie instrukcji `INSERT` prezentowanych w tym podrozdziale.

Jeżeli wstawimy nowy wiersz do tabeli `order_status`, ale nie określmy wartości dla kolumn `status` i `last_modified`, zostaną w nich umieszczone wartości domyślne. Na przykład w poniżej instrukcji `INSERT` pominięto wartości dla kolumn `status` i `last_modified`:

```
INSERT INTO order_status (order_status_id)
VALUES (1);
```

W kolumnie `status` zostanie umieszczona domyślna wartość 'Zamówienie złożone', a w kolumnie `last_modified` zostanie zapisana bieżąca data i godzina.

Wartości domyślne możemy nadpisać, przesyłając wartości dla kolumn, co obrazuje poniższy przykład:

```
INSERT INTO order_status (
    order_status_id, status, last_modified
) VALUES (
    2, 'Zamówienie wysłane', '04/06/10'
);
```

To zapytanie pobiera wiersze z tabeli `order_status`:

```
SELECT *
FROM order_status
```

ORDER_STATUS_ID	STATUS	LAST_MOD
1	Zamówienie złożone	11/10/07
2	Zamówienie wysłane	04/06/10

Możemy przywrócić w kolumnie wartość domyślną, używając słowa kluczowego `DEFAULT` w instrukcji `UPDATE`. Na przykład poniższa instrukcja `UPDATE` wprowadza wartość domyślną do kolumny `status`:

```
UPDATE order_status
SET status = DEFAULT
WHERE order_status_id = 2;
```

To zapytanie obrazuje zmiany dokonane przez powyższą instrukcję `UPDATE`:

```
SELECT *
FROM order_status
```

ORDER_STATUS_ID	STATUS	LAST_MOD
1	Zamówienie złożone	11/10/07
2	Zamówienie złożone	04/06/10

Scalanie wierszy za pomocą instrukcji MERGE

Instrukcja `MERGE` umożliwia aktualizację i dołączanie wierszy z jednej tabeli do drugiej. Możemy na przykład chcieć złączyć z tabelą `products` zmiany wprowadzone w produktach w innej tabeli.

Schemat `store` zawiera tabelę o nazwie `product_changes`, która została utworzona przez skrypt `store_schema.sql` za pomocą następującej instrukcji `CREATE TABLE`:

```
CREATE TABLE product_changes (
    product_id INTEGER
        CONSTRAINT prod_changes_pk PRIMARY KEY,
    product_type_id INTEGER
        CONSTRAINT prod_changes_fk_product_types
            REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);
```

Poniższe zapytanie pobiera z tej tabeli `product_changes` kolumny `product_id`, `product_type_id`, `name` i `price`:

```
SELECT product_id, product_type_id, name, price
FROM product_changes;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Nauka współczesna	40
2	1	Nowa chemia	35

3	1 Supernowa	25,99
13	2 Lądownie na Księżyku	15,99
14	2 Łódź podwodna	15,99
15	2 Samolot	15,99

Założymy, że chcemy skalić wiersze z tabeli `product_changes` z tabelą `products` zgodnie z poniższymi regułami:

- W przypadku wierszy z takimi samymi wartościami `product_id` w obu tabelach do istniejących wierszy tabeli `products` powinny zostać wpisane wartości kolumn z tabeli `product_changes`. Na przykład w tabeli `product_changes` cena produktu nr 1 jest inna niż w tabeli `products`, dlatego też cena tego produktu musi zostać zmieniona w tabeli `products`. Produkt nr 2 ma inną nazwę i cenę, więc w tabeli `products` muszą zostać zmienione obydwie wartości, a produkt nr 3 ma inną wartość `product_type_id`, więc w tabeli `products` musi ona zostać zmieniona.
 - Wiersze występujące tylko w tabeli `product_changes` powinny zostać wstawione do tabeli `products`. Produkty nr 13, 14 i 15 występują tylko w tabeli `product_changes`, muszą więc zostać wstawione do tabeli `products`.

Instrukcję **MERGE** najlepiej zrozumieć, analizując przykład (poniższa wykona scalenie zgodnie z przedstawioną listą reguł):

```

MERGE INTO products p
USING product_changes pc ON (
    p.product_id = pc.product_id
)
WHEN MATCHED THEN
    UPDATE
        SET
            p.product_type_id = pc.product_type_id,
            p.name = pc.name,
            p.description = pc.description,
            p.price = pc.price
WHEN NOT MATCHED THEN
    INSERT (
        p.product_id, p.product_type_id, p.name,
        p.description, p.price
    ) VALUES (
        pc.product_id, pc.product_type_id, pc.name,
        pc.description, pc.price
    );

```

6 rows merged.

W katalogu SQL znajduje się skrypt `merge_example.sql`. Skrypt ten zawiera powyższą instrukcję MERGE i pozwala wykonać zaprezentowany przykład bez konieczności ręcznego przepisywania.

W instrukcji **MERGE** są następujące części:

- Klauzula **INTO** określa nazwę tabeli, z którą będą scalane wiersze. W przykładzie jest to tabela **products**, której nadano alias **p**.
 - Klauzula **USING ... ON** określałą złączenie tabeli. W przykładzie złączenie jest dokonywane przez kolumny **product_id** tabel **products** i **product_changes**. Tabeli **product_changes** nadano alias **pc**.
 - Klauzula **WHEN MATCHED THEN** określa czynność wykonywaną, gdy wiersz spełnia klauzulę **USING ... ON**. W przykładzie czynnością jest instrukcja **UPDATE**, która przypisuje wartościom kolumn **product_type_id**, **name**, **description** i **price** wiersza istniejącego w tabeli **products** wartości odpowiedniego wiersza z tabeli **product_changes**.
 - Klauzula **WHEN NOT MATCHED THEN** określa czynność wykonywaną, gdy wiersz *nie* spełnia klauzuli **USING ... ON**. W przykładzie czynnością jest instrukcja **INSERT**, która wstawia wiersz do tabeli **products**, pobierając wartości kolumn z wiersza z tabeli **product_changes**.

Po uruchomieniu przedstawionej instrukcji MERGE widzimy, że zostało scalonych sześć wierszy. Są to te, w których `product_id` ma wartość 1, 2, 3, 13, 14 i 15. Poniższe zapytanie pobiera z tabeli `products` sześć scalonych wierszy:

```
SELECT product_id, product_type_id, name, price
FROM products
WHERE product_id IN (1, 2, 3, 13, 14, 15);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Nauka współczesna	40
2	1	Nowa chemia	35
3	1	Supernowa	25,99
13	2	Lądownie na Księżyco	15,99
14	2	Tłoź podwodna	15,99
15	2	Samolot	15,99

W tych wierszach zostały wprowadzone następujące zmiany:

- produkt nr 1 ma nową cenę,
- produkt nr 2 ma nową nazwę i cenę,
- produkt nr 3 ma nowy identyfikator rodzaju produktu (`product_type_id`),
- produkty nr 13, 14 i 15 zostały dodane do tabeli.

Wiemy już, jak możemy zmieniać zawartość tabel, zajmijmy się więc transakcjami.

Transakcje bazodanowe

Transakcja bazodanowa jest grupą instrukcji SQL, która wykonuje **logiczną jednostkę pracy**. Możemy ją sobie wyobrazić jako nierozerłączny zbiór instrukcji SQL, których wyniki powinny być zapisane w bazie danych w całości (lub zostać w całości wycofane).

Przykładem transakcji bazodanowych jest przelew pieniędzy z konta bankowego na inne. Instrukcja `UPDATE` odejmuję jednak daną sumę od ilości gotówki na jednym koncie, a druga instrukcja `UPDATE` dodaje ją do pieniędzy na drugim koncie. Zarówno odejmowanie, jak i dodawanie muszą być trwale zapisane w bazie danych. W przeciwnym razie pieniądze zostaną utracone. Jeżeli wystąpi problem z przelewem, wówczas zarówno odejmowanie, jak i dodawanie muszą zostać wycofane. W tym prostym przykładzie są wykorzystywane jedynie dwie instrukcje `UPDATE`, transakcja może jednak zawierać wiele instrukcji `INSERT`, `UPDATE` i `DELETE`.

Zatwierdzanie i wycofywanie transakcji

Aby trwale zapisać wyniki instrukcji SQL znajdujących się w transakcji, należy wykonać **zatwierdzenie** za pomocą instrukcji SQL `COMMIT`. Jeżeli chcemy **wycofać wyniki**, wykonujemy instrukcję SQL `ROLLBACK`, która przywraca pierwotne wartości wierszy.

W poniższym przykładzie do tabeli `customers` jest dodawany jeden wiersz, a następnie zmiana jest utrwalana przez wykonanie instrukcji `COMMIT`:

```
INSERT INTO customers
VALUES (6, 'Fryderyk', 'Zielony', '70/01/01', '800-555-1215');
```

1 row created.

```
COMMIT;
```

Commit complete.

W poniższym przykładzie zmieniane są wartości dla klienta nr 1, a następnie zmiany te są wycofywane za pomocą instrukcji `ROLLBACK`:

```
UPDATE customers
SET first_name = 'Edward'
WHERE customer_id = 1;
```

1 row updated.

ROLLBACK;

Rollback complete.

To zapytanie pokazuje nowy wiersz zatwierdzony przez instrukcję **COMMIT**:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	
6	Fryderyk	Zielony	70/01/01	800-555-1215

Informacje o kliencie nr 6 zostały utrwalone przez instrukcję **COMMIT**, a zmiany wprowadzone do informacji o tym kliencie zostały wycofane przez instrukcję **ROLLBACK**.

Rozpoczynanie i kończenie transakcji

Transakcja jest logiczną jednostką pracy, która umożliwia rozdzielenie instrukcji SQL. Posiada początek i koniec.

Transakcja rozpoczyna się, gdy ma miejsce jedno ze zdarzeń:

- połączenie z bazą danych i wykonanie instrukcji DML (**INSERT**, **UPDATE** lub **DELETE**),
- wcześniejsza transakcja kończy się i wprowadzamy kolejną instrukcję DML.

Transakcja kończy się, gdy:

- Zostaną wykonane instrukcje **COMMIT** lub **ROLLBACK**.
- Zostanie wykonana instrukcja DDL, taka jak **CREATE TABLE** — w takim przypadku zatwierdzenie jest dokonywane automatycznie.
- Zostanie wykonana instrukcja DCL, taka jak **GRANT** — w takim przypadku zatwierdzenie jest dokonywane automatycznie. Instrukcja **GRANT** zostanie opisana w następnym rozdziale.
- Nastąpi rozłączenie z bazą danych. Jeżeli zakończymy pracę programu SQL*Plus w zwykły sposób — za pomocą polecenia **EXIT** — automatycznie zostanie wykonana instrukcja **COMMIT**. Jeżeli program SQL*Plus zakończy pracę w niestandardowy sposób — na przykład jeżeli wystąpi awaria w systemie, w którym program został uruchomiony — automatycznie zostanie wykonana instrukcja **ROLLBACK**. Te reguły są stosowane do wszystkich programów uzyskujących dostęp do bazy danych. Jeśli na przykład napiszemy w Javie program, który połączył się z bazą danych i uległ awarii, automatycznie zostanie wykonana instrukcja **ROLLBACK**.
- Wykonywanie instrukcji DML nie powiedzie się — w tym przypadku instrukcja **ROLLBACK** jest wykonywana jedynie dla tej konkretnej instrukcji DML.



Niezatwierdzanie lub niewycofywanie transakcji w sposób jawny jest złą praktyką — zawsze na końcu transakcji należy wykonać instrukcję **COMMIT** lub **ROLLBACK**.

Punkty zachowania

W dowolnym miejscu transakcji możemy umieścić **punkt zachowania**. Umożliwia to wycofanie zmian aż do tego punktu. Punkty zachowania są przydatne przy dzieleniu bardzo długich transakcji, ponieważ

gdy popełnimy błąd po ustawieniu ich, nie musimy wycofywać całej transakcji. Należy z nich jednak korzystać rozważnie — lepszym rozwiążaniem może być rozbicie transakcji na mniejsze.

Przykład zastosowania punktu zachowania zostanie przedstawiony wkrótce, a na razie sprawdźmy ceny produktów nr 4 i 5:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	13,95
5	49,99

Cena produktu nr 4 wynosi 13,95 zł, a produktu nr 5 — 49,99 zł.

Poniższa instrukcja UPDATE zwiększa cenę produktu nr 4 o 20 procent:

```
UPDATE products
SET price = price * 1.20
WHERE product_id = 4;
```

1 row updated.

Poniższa instrukcja ustawia punkt zachowania o nazwie `save1`:

```
SAVEPOINT save1;
```

Savepoint created.

Od teraz wszystkie kolejne instrukcje DML mogą zostać wycofane do tego punktu zachowania, a zmiana dokonana we wpisie dotyczącym produktu nr 4 zostanie zachowana.

Poniższa instrukcja UPDATE zwiększa cenę produktu nr 5 o 30 procent:

```
UPDATE products
SET price = price * 1.30
WHERE product_id = 5;
```

1 row updated.

Poniższe zapytanie pobiera ceny obu produktów:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	16,74
5	64,99

Cena produktu nr 4 wzrosła o 20 procent, a produktu nr 5 — o 30 procent.

Poniższa instrukcja wycofuje transakcję do wcześniej ustawionego punktu zachowania:

```
ROLLBACK TO SAVEPOINT save1;
```

Rollback complete.

W ten sposób została wycofana zmiana ceny produktu nr 5, ale pozostała zmiana ceny produktu nr 4, co obrazuje poniższe zapytanie:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	16,74
5	49,99

Zgodnie z oczekiwaniami cena produktu nr 4 jest wciąż wyższa, cena produktu nr 5 powróciła natomiast do pierwotnej wartości.

Instrukcja **ROLLBACK** wycofuje całą transakcję:

```
ROLLBACK;
```

Rollback complete.

Zmiana ceny produktu nr 4 została wycofana, co obrazuje poniższe zapytanie:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	13,95
5	49,99

ACID — właściwości transakcji

Wcześniej zdefiniowaliśmy transakcję jako **logiczną jednostkę pracy**, czyli grupę powiązanych z sobą instrukcji SQL, które są *zatwierdzane* lub *wycofywane* jako jedna jednostka. Bardziej restrykcyjna definicja transakcji z teorii baz danych stwierdza, że transakcja ma cztery podstawowe właściwości zwane **ACID** (od pierwszych liter ich angielskich nazw):

- **Atomic** (atomowość) — wszystkie instrukcje SQL w transakcji stanowią jedną jednostkę pracy.
- **Consistent** (spójność) — transakcje zapewniają, że stan bazy danych pozostaje spójny, co oznacza, że baza danych przy rozpoczęciu transakcji znajduje się w stanie spójnym, a po zakończeniu transakcji znajduje się w innym, też spójnym stanie.
- **Isolated** (izolacja) — oddzielne transakcje nie powinny na siebie wpływać.
- **Durable** (trwałość danych) — po zatwierdzeniu transakcji zmiany w bazie danych są utrwalane, nawet jeśli komputer, na którym zostało uruchomione oprogramowanie bazy danych, ulegnie awarii.

Oprogramowanie bazy danych Oracle obsługuje właściwości ACID i daje duże możliwości przywracania bazy danych po awarii systemu.

Transakcje współbieżne

Oprogramowanie baz danych Oracle obsługuje interakcje wielu użytkowników z bazą danych, z których każdy w tym samym czasie może wykonywać własne transakcje. Takie transakcje nazywane są **współbieżnymi**.

Jeżeli użytkownicy uruchamiają transakcje mające wpływ na tę samą tabelę, ich efekty są od siebie oddzielone, dopóki nie zostanie wykonana instrukcja **COMMIT**. Poniższa sekwencja zdarzeń oparta na dwóch transakcjach (T1 i T2), pracujących na tabeli **customers**, obrazuje rozdzielnosć transakcji:

1. T1 i T2 wykonują instrukcję **SELECT**, pobierającą wszystkie wiersze z tabeli **customers**.
2. T1 wykonuje instrukcję **INSERT**, wstawiającą wiersz do tabeli **customers**, nie wykonuje jednak jeszcze instrukcji **COMMIT**.
3. T2 wykonuje kolejną instrukcję **SELECT** i pobiera te same wiersze co w kroku 1. T2 nie „widzi” wiersza dodanego przez T1 w kroku 2.
4. T1 wykonuje w końcu instrukcję **COMMIT**, aby trwale zapisać nowy wiersz wstawiony w kroku 2.
5. T2 wykonuje kolejną instrukcję **SELECT** i w końcu „widzi” nowy wiersz wstawiony przez T1.

Podsumowując: T2 nie widzi zmian dokonanych przez T1 do momentu, aż T1 nie zatwierdzi wprowadzonych zmian. To jest domyślny poziom izolacji transakcji, jak jednak dowiesz się z podrozdziału „Poziomy izolacji transakcji”, poziom izolacji można zmieniać.

Tabela 9.1 zawiera przykładowe instrukcje SQL obrazujące działanie współbieżnych transakcji. Zaprezentowano w niej przemienną kolejność wykonywania instrukcji przez transakcje T1 i T2. T1 pobiera wiersze, wstawia wiersz i modyfikuje wiersz w tabeli *customers*. T2 pobiera wiersze z tabeli *customers*. Nie widzi zmian dokonanych przez T1 do momentu, gdy T1 ich nie zatwierdzi. Można wprowadzić instrukcje przedstawione w tabeli 9.1 i zobaczyć ich wyniki, uruchamiając dwie sesje SQL*Plus i logując się w obu jako użytkownik *store*. Instrukcje należy wpisywać w naprzemiennej kolejności przedstawionej w tabeli 9.1.

Tabela 9.1. Transakcje współbieżne

Transakcja 1 T1	Transakcja 2 T2
(1) SELECT * FROM customers;	(2) SELECT * FROM customers;
(3) INSERT INTO customers (customer_id, first_name, last_name) VALUES (7, 'Jan', 'Cena');	
(4) UPDATE customers SET last_name = 'Pomarańcz' WHERE customer_id = 2;	(6) SELECT * FROM customers;
Zwrócony zestaw wyników zawiera nowy wiersz i modyfikację	Zwrócony zestaw wyników nie zawiera nowego wiersza i zmiany wprowadzonej przez T1. Zestaw wyników zawiera pierwotne wiersze pobrane w kroku 2.
(7) COMMIT; Wstawienie nowego wiersza i modyfikacja zostają zatwierdzone	(8) SELECT * FROM customers; Zestaw wyników zawiera nowy wiersz i modyfikację wprowadzone przez T1 w krokach 3. i 4.

Blokowanie transakcji

W celu obsłużenia transakcji współbieżnych oprogramowanie bazy danych Oracle musi zapewnić poprawność danych w tabelach. Służą do tego **blokady**. Rozważmy przykład, w którym dwie transakcje o nazwach T1 i T2 próbują zmodyfikować w tabeli *customers* informacje o kliencie nr 1:

1. T1 wykonuje instrukcję **UPDATE** w celu zmodyfikowania informacji o kliencie nr 1, ale nie wykonuje instrukcji **COMMIT**. W takiej sytuacji mówimy, że transakcja T1 „zablokowała” wiersz.
2. T2 również próbuje wykonać instrukcję **UPDATE** w celu zmodyfikowania informacji o kliencie nr 1, nie uzyska jednak blokady wiersza, ponieważ jest on już zablokowany przez T2. Instrukcja **UPDATE** z transakcji T2 musi poczekać, aż T1 zakończy się i zwolni blokadę wiersza.
3. T1 kończy się przez wykonanie **COMMIT**, tym samym zostaje zwolniona blokada wiersza.
4. T2 uzyskuje blokadę wiersza i jest wykonywana instrukcja **UPDATE**. T2 przetrzymuje blokadę wiersza aż do zakończenia całej transakcji.

Podsumowując: transakcja nie może uzyskać blokady wiersza, jeżeli w tym czasie jest ona przetrzymywana przez inną transakcję.



Domyślne zasady blokowania najłatwiej przedstawić w następujący sposób: instrukcje czytające nie blokują innych instrukcji czytających, instrukcje zapisujące nie blokują instrukcji czytających, a instrukcje zapisujące blokują inne instrukcje zapisujące, jeżeli próbują zmodyfikować ten sam wiersz.

Poziomy izolacji transakcji

Poziom izolacji transakcji jest stopniem, do jakiego zmiany dokonywane przez jedną transakcję są odzielone od innej transakcji wykonywanej współbieżnie. Zanim omówię dostępne poziomy izolacji, musisz poznać rodzaje problemów, jakie mogą wystąpić, gdy współbieżne transakcje próbują uzyskać dostęp do tego samego wiersza tabeli.

Można wyróżnić trzy typy potencjalnych problemów związanych z przetwarzaniem transakcji w sytuacji, gdy transakcje współbieżne odwołują się do tych samych wierszy:

- **Fantomowy odczyt** — T1 odczytuje zestaw wierszy zwrócony przez przeslaną klauzulę WHERE. T2 wstawia później nowy wiersz spełniający warunek z klauzuli WHERE, zastosowany w transakcji T1. Następnie T1 ponownie odczytuje wiersze, tym razem widzi jednak dodatkowy wiersz wstawiony przez T2. Ten nowy wiersz nazywamy fantomem, ponieważ z perspektywy transakcji T1 pojawił się on w niewyjaśniony sposób.
- **Niepowtarzalny odczyt** — T1 czyta wiersz, a T2 modyfikuje wiersz właśnie odczytany przez T1. Następnie T1 ponownie odczytuje ten wiersz i odkrywa, że się zmienił. Taką sytuację nazywamy niepowtarzalnym odczytem, ponieważ wiersz pierwotnie odczytany przez T1 został zmieniony.
- **Brudny odczyt** — T1 modyfikuje wiersz, ale nie zatwierda zmiany, po czym T2 czyta zmodyfikowany wiersz. Następnie T1 wycofuje poprzednią modyfikację. Teraz wiersz właśnie przeczytany przez T2 jest już nieprawidłowy (jest „brudny”), ponieważ aktualizacja dokonana przez T1 nie została zatwierdzona, gdy wiersz został odczytany przez T2.

W celu rozwiązywania tych problemów w systemach baz danych zaimplementowano różne poziomy izolacji, pozwalając tym samym zapobiec wpływu na siebie współbieżnych transakcji. W standardzie SQL zdefiniowano następujące poziomy izolacji, prezentowane poniżej w kolejności rosnącej:

- **READ UNCOMMITTED** — dopuszcza fantomy, niepowtarzalne odczyty i brudne odczyty.
- **READ COMMITTED** — dopuszcza fantomy i niepowtarzalne odczyty, ale nie dopuszcza do brudnych odczytów.
- **REPEATABLE READ** — dopuszcza fantomy, ale nie dopuszcza do odczytów niepowtarzalnych i brudnych.
- **SERIALIZABLE** — nie zezwala na fantomy, niepowtarzalne odczyty i brudne odczyty.

Oprogramowanie bazy danych Oracle obsługuje poziomy izolacji transakcji **READ COMMITTED** i **SERIALIZABLE**. Nie obsługuje poziomów **READ UNCOMMITTED** i **REPEATABLE READ**.

W standardzie SQL domyślnym poziomem izolacji transakcji jest **SERIALIZABLE**, w bazie danych Oracle domyślnym poziomem izolacji jest natomiast **READ COMMITTED**, który jest wystarczający w niemal wszystkich zastosowaniach.

 **Ostrzeżenie** Choć w bazie danych Oracle można używać poziomu **SERIALIZABLE**, czas przetwarzania instrukcji SQL może się znaczco wydłużyć. Poziom ten powinien być wykorzystywany tylko wtedy, gdy jest to naprawdę niezbędne.

Poziom izolacji transakcji jest ustawiany za pomocą instrukcji **SET TRANSACTION**. Na przykład poniższa instrukcja ustawia poziom izolacji transakcji **SERIALIZABLE**:

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

Omówię teraz przykład transakcji, w której jest wykorzystywany ten poziom izolacji.

Przykład transakcji **SERIALIZABLE**

W tym podrozdziale zostanie przedstawiony przykład prezentujący efekt ustawienia poziomu izolacji transakcji na **SERIALIZABLE**.

W przykładzie wykorzystano dwie transakcje, T1 i T2. T1 ma domyślny poziom izolacji, czyli **READ COMMITTED**, a T2 ma poziom izolacji **SERIALIZABLE**. T1 i T2 odczytują wiersze z tabeli *customers*, a następnie T1 wstawia nowy i modyfikuje istniejący wiersz w tej tabeli. Ponieważ transakcja T2 ma poziom **SERIALIZABLE**, nie widzi wstawionego wiersza ani zmiany wprowadzonej przez T1, *nawet po tym*, jak T1

zatwierdzi zmiany. Wynika to stąd, że odczyt wstawionego wiersza byłby fantomem, odczyt zmodyfikowanego wiersza byłby natomiast niepowtarzalny, a takie operacje nie są dozwolone na tym poziomie.

Tabela 9.2 przedstawia instrukcje SQL składające się na transakcje T1 i T2, w kolejności ich wykonywania.

Tabela 9.2. Transakcje SERIALIZABLE

Transakcja 1 T1 (READ COMMITTED)	Transakcja 2 T2 (SERIALIZABLE)
	(1) SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
(3) SELECT * FROM customers;	(2) SELECT * FROM customers;
(4) INSERT INTO customers (customer_id, first_name, last_name) VALUES (8, 'Stefan', 'Guzik');	
(5) UPDATE customers SET last_name = 'Zółty' WHERE customer_id = 3;	
(6) COMMIT;	(8) SELECT * FROM customers;
(7) SELECT * FROM customers;	Zwrócony zestaw wyników wcijaż nie zawiera nowego wiersza i zmiany wprowadzonej przez T1, ponieważ T2 ma poziom SERIALIZABLE
Zwrócony zestaw wyników zawiera nowy wiersz oraz wprowadzoną zmianę	

Wydanie polecenia kończącego transakcję SERIALIZABLE zsynchronizuje transakcję SERIALIZABLE z transakcją READ COMMITTED. Na przykład wydanie polecenia COMMIT w transakcji SERIALIZABLE, nawet jeśli żadne wiersze nie zostały zmodyfikowane przez transakcję SERIALIZABLE, zsynchronizuje tę transakcję z transakcją READ COMMITTED.

Zapytania retrospektywne

Jeżeli przypadkiem zatwierdzimy zmiany i będziemy chcieli przywrócić pierwotną postać wierszy, możemy użyć zapytania retrospektynego. Następnie możemy na podstawie wyników tego zapytania samodzielnie przywrócić pierwotne wartości wierszy.

Zapytania retrospektywne mogą opierać się na dacie i godzinie lub numerze SCN (ang. *system change number* — systemowym numerze zmiany). Oprogramowanie bazy danych wykorzystuje numery SCN do śledzenia zmian danych i możemy je wykorzystać do przejrzenia określonego SCN w bazie danych.



Aby upewnić się, że otrzymasz po wykonaniu instrukcji zaprezentowanych w tym podrozdziale wyniki takie jak w zaprezentowanych przykładach, należy ponownie uruchomić skrypt *store_schema.sql*.

Przyznawanie uprawnień do używania zapytań retrospektywnych

Zapytania retrospektywne wykorzystują pakiet DBMS_FLASHBACK PL/SQL, do uruchomienia którego jest wymagane uprawnienie EXECUTE. W poniższym przykładzie następuje połączenie z bazą danych użytkownika sys i przyznanie użytkownikowi store uprawnienia EXECUTE dla pakietu DBMS_FLASHBACK:

```
CONNECT sys/wprowadzone_przy_instalacji AS sysdba
GRANT EXECUTE ON SYS.DBMS_FLASHBACK TO store;
```



Jeżeli nie jest możliwe wykonanie tych instrukcji, należy skontaktować się z administratorem bazy danych. Uprawnienia zostaną szczegółowo opisane w następnym rozdziale, a pakiety PL/SQL — w rozdziale 12.

Zapytania retrospektywne w oparciu o czas

W poniższym przykładzie następuje połączenie z bazą danych użytkownika `store` i pobranie kolumn `product_id`, `name` i `price` pierwszych pięciu produktów z tabeli `products`:

```
CONNECT store/store_password
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	19,95
2	Chemia	30
3	Supernowa	25,99
4	Wojny czołgów	13,95
5	Z Files	49,99

W kolejnym przykładzie następuje zmniejszenie cen w tych wierszach, zatwierdzenie zmiany oraz ponowne pobranie wierszy w celu sprawdzenia modyfikacji:

```
UPDATE products
SET price = price * 0.75
WHERE product_id <= 5;

COMMIT;
```

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	14,96
2	Chemia	22,5
3	Supernowa	19,49
4	Wojny czołgów	10,46
5	Z Files	37,49

Poniższa instrukcja uruchamia procedurę `DBMS_FLASHBACK.ENABLE_AT_TIME()`, która umożliwia przejście retrospektywne do konkretnej daty i godziny. Należy zauważyć, że procedura ta przyjmuje wyrażenie typu *DataGodzina*, a w przykładzie jest przesyłane wyrażenie `SYSDATE - 10 / 1440` (po obliczeniu to wyrażenie oznacza „dziesięć minut temu”):

```
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 10 / 1440);
```



24 godziny×60 minut w godzinie = 1440 minut. Dlatego też `SYSDATE - 10 / 1440` oznacza dziesięć minut temu.

Wszystkie uruchamiane zapytania będą teraz zwracały wiersze w takiej postaci, w jakiej były 10 minut temu. Z zakładając, że wykonaliśmy powyższą instrukcję `UPDATE` nie wcześniej niż 10 minut temu, późniejsze zapytanie zwróci takie ceny, jakie były zapisane przed dokonaniem zmiany:

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	19,95
2	Chemia	30
3	Supernowa	25,99
4	Wojny czołgów	13,95
5	Z Files	49,99

W celu wyłączenia retrospektywnej uruchamiamy procedurę DBMS_FLASHBACK.DISABLE(), tak jak poniżej:

```
EXECUTE DBMS_FLASHBACK.DISABLE();
```



Retrospektywę należy wyłączyć przed jej ponownym włączeniem.

Ostrzeżenie
Teraz, gdy wykonamy zapytanie, zostaną pobrane bieżące wartości wierszy, co obrazuje poniższy przykład:

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	14,96
2	Chemia	22,5
3	Supernowa	19,49
4	Wojny czołgów	10,46
5	Z Files	37,49

Zapytania retrospektywne z użyciem SCN

Retrospektywne oparte na systemowych numerach zmian (SCN) mogą być bardziej precyzyjne niż te bazujące na czasie, ponieważ baza danych wykorzystuje SCN do śledzenia zmian wprowadzanych do danych. Aby pobrać bieżący SCN, możemy uruchomić procedurę DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER():

```
VARIABLE current_scn NUMBER

EXECUTE :current_scn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();

PRINT current_scn
```

CURRENT_SCN
1682048

W kolejnym przykładzie do tabeli products jest dodawany wiersz, następnie zatwierdzenie zmiany, a następnie zostaje pobrany nowy wiersz:

```
INSERT INTO products (
    product_id, product_type_id, name, description, price
) VALUES (
    15, 1, 'Fizyka', 'Podręcznik fizyki', 39.95
);

COMMIT;
```

```
SELECT *
FROM products
WHERE product_id = 15;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
15	1	Fizyka

DESCRIPTION	PRICE
Podręcznik fizyki	39,95

W kolejnym przykładzie jest uruchamiana procedura DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(), która umożliwia wykonanie retrospektywnej do SCN. Należy zauważyć, że procedura przyjmuje SCN i w przykładzie przesłano do niej zmienną current_scn:

```
EXECUTE DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(:current_scn);
```

Odtąd wszystkie uruchamiane zapytania będą wyświetlały wiersze w postaci, w jakiej znajdowały się przy SCN zapisanym w `current_scn`, czyli przed wykonaniem instrukcji `INSERT`. Poniższe zapytanie próbuje pobrać wiersz, w którym `product_id` ma wartość 15. Próba kończy się niepowodzeniem, ponieważ wiersz został dodany już po zapisaniu SCN w zmiennej `current_scn`:

```
SELECT product_id  
FROM products  
WHERE product_id = 15;  
  
no rows selected
```

W celu wyłączenia retrospektywnej należałoby uruchomić procedurę `DBMS_FLASHBACK.DISABLE()`:

```
EXECUTE DBMS_FLASHBACK.DISABLE();
```

Jeżeli ponownie wykonamy wcześniej przedstawione zapytanie, zostanie zwrócony wiersz dodany przez instrukcję `INSERT`:

```
SELECT product_id  
FROM products  
WHERE product_id = 15;  
  
PRODUCT_ID  
-----  
15
```

Podsumowanie

Z tego rozdziału dowiedziałeś się:

- jak wstawiać wiersze za pomocą instrukcji `INSERT`,
- jak modyfikować wiersze za pomocą instrukcji `UPDATE`,
- jak usuwać wiersze za pomocą instrukcji `DELETE`,
- w jaki sposób baza danych zachowuje integralność przez wymuszanie więzów,
- jak używać słowa kluczowego `DEFAULT` do określania domyślnej wartości kolumny,
- jak za pomocą instrukcji `MERGE` aktualizować wiersze jednej tabeli danymi z innej tabeli,
- że transakcja jest grupą instrukcji SQL, stanowiącą logiczną jednostkę pracy,
- że oprogramowanie bazy danych Oracle może obsługiwać wiele współbieżących transakcji,
- jak używać zapytań retrospektywnych do przeglądania zawartości wierszy przed wprowadzeniem do nich zmian.

W następnym rozdziale zawarto informacje o użytkownikach, uprawnieniach i rolach.

ROZDZIAŁ

10

Użytkownicy, uprawnienia i role

Z tego rozdziału dowiesz się:

- jak zarządzać kontami użytkowników,
- jakie uprawnienia umożliwiają użytkownikom wykonywanie zadań w bazie danych,
- o systemowych i obiektowych typach uprawnień,
- w jaki sposób uprawnienia systemowe umożliwiają wykonywanie takich działań jak uruchamianie instrukcji DDL,
- jak uprawnienia obiektowe umożliwiają wykonywanie takich działań jak uruchamianie instrukcji DML,
- jak grupować uprawnienia w role,
- jak obserwować wykonywanie instrukcji SQL.



Przed uruchomieniem przykładów z tego rozdziału należy ponownie uruchomić skrypt `store_schema.sql`. Jeżeli chcesz wykonać przykłady zamieszczone w tym rozdziale, musisz samodzielnie wpisywać instrukcje SQL, ponieważ nie znajdują się one w żadnym skrypcie.

Bardzo krótkie wprowadzenie do przechowywania danych

W tym podrozdziale znajduje się bardzo krótkie wprowadzenie do złożonego i ogromnego tematu przechowywania danych w bazach danych. Musisz zrozumieć terminologię zaprezentowaną w tym wprowadzeniu.

W tym rozdziale napotkamy określenie **przestrzeń tabel**. Baza danych wykorzystuje przestrzenie tabel do składowania odrębnych obiektów, takich jak tabele, typy, kod PL/SQL itd. Powiązane z sobą obiekty zwykle są grupowane i składowane w tej samej przestrzeni tabel. Możemy na przykład utworzyć aplikację do wprowadzania zamówień i składować wszystkie obiekty wykorzystywane przez nią w jednej przestrzeni tabel. Możemy także utworzyć aplikację do obsługi łańcucha dostaw i składować obiekty wykorzystywane przez nią w innej przestrzeni tabel. Użytkownikowi możemy przydzielić miejsce w przestrzeni tabel do przechowywania jego danych.

Przestrzenie tabel są zapisywane w plikach danych, które są plikami przechowywanymi w systemie plików serwera bazy danych. Plik danych przechowuje dane jednej przestrzeni tabel. Oprogramowanie

bazy danych Oracle tworzy plik danych dla przestrzeni tabel, alokując wymaganą ilość miejsca na dysku z dodatkowym miejscem na nagłówek pliku.

Więcej informacji na temat przestrzeni tabel znajduje się w podręczniku *Oracle Database Concepts* opublikowanym przez Oracle Corporation.

Użytkownicy

W tym podrozdziale nauczysz się tworzyć konto użytkownika i usuwać je oraz zmieniać hasło dostępu do niego.

Tworzenie konta użytkownika

Do tworzenia konta użytkownika w bazie danych służy instrukcja `CREATE USER`. Jej uproszczona składnia ma następującą postać:

```
CREATE USER nazwa_użytkownika IDENTIFIED BY hasło
[DEFAULT TABLESPACE domyślna_przestrzeń_tabel]
[TEMPORARY TABLESPACE tymczasowa_przestrzeń_tabel];
```

gdzie:

- *nazwa_użytkownika* jest nazwą użytkownika bazy danych.
- *hasło* jest hasłem użytkownika bazy danych.
- *domyślna_przestrzeń_tabel* jest domyślną przestrzenią tabel, w której będą składowane obiekty bazy danych. Jeżeli pominiemy tę część, zostanie użyta domyślna przestrzeń tabel `SYSTEM`, która zawsze istnieje w bazie danych.
- *tymczasowa_przestrzeń_tabel* jest domyślną przestrzenią tabel, w której będą składowane tymczasowe obiekty, między innymi tabele tymczasowe, które zostaną opisane w następnym rozdziale. Jeżeli nie określmy tymczasowej przestrzeni tabel, zostanie użyta domyślna przestrzeń tabel `SYSTEM`.

W poniższym przykładzie łączymy się z bazą danych jako użytkownik `system` i tworzymy konto użytkownika `roman` z hasłem `cena`:

```
CONNECT system/oracle
CREATE USER roman IDENTIFIED BY cena;
```



Do wykonania prezentowanych przykładów konieczne są podwyższone uprawnienia. W powyższym przykładzie zostało użyte konto użytkownika `system`, którego hasłem w zainstalowanej bazie danych jest `oracle`.

W kolejnym przykładzie jest tworzone konto użytkownika `henryk` i jest określana domyślna oraz tymczasowa przestrzeń tabel:

```
CREATE USER henryk IDENTIFIED BY hora
DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
```

Jeżeli w bazie danych nie istnieją przestrzenie tabel o nazwach `users` i `temp`, można pominąć powyższy przykład. Konto użytkownika `henryk` nie jest wykorzystywane w dalszej części książki, a przykład został zamieszczony jedynie po to, aby zaprezentować sposób określania przestrzeni tabel. Wszystkie przestrzenie tabel w bazie danych można wyświetlić, łącząc się jako użytkownik `system` i uruchamiając zapytanie `SELECT tablespace_name FROM dba_tablespaces`.

Jeżeli chcemy, aby użytkownik mógł wykonywać jakieś czynności w bazie danych, musimy mu nadać odpowiednie uprawnienia. Na przykład w celu połączenia się z bazą danych użytkownik musi być uprawniony do tworzenia sesji, czyli musi mieć uprawnienie systemowe `CREATE SESSION`. Uprawnienia są przyznawane przez uprzywilejowanych użytkowników (na przykład `system`) za pomocą instrukcji `GRANT`.

W poniższym przykładzie użytkownikowi `roman` nadawane jest uprawnienie `CREATE SESSION`:

```
GRANT CREATE SESSION TO roman;
```

Użytkownik **roman** będzie mógł teraz połączyć się z bazą danych.

W poniższym przykładzie zostały utworzone konta pozostałych użytkowników, wykorzystywane w tym rozdziale, i nadano im uprawnienia CREATE SESSION:

```
CREATE USER stefan IDENTIFIED BY guzik;
CREATE USER gabi IDENTIFIED BY tort;
GRANT CREATE SESSION TO stefan, gabi;
```

Zmianianie hasła użytkownika

Hasło użytkownika można zmienić za pomocą instrukcji ALTER USER. Na przykład poniższa instrukcja zmienia hasło użytkownika **roman** na **kora**:

```
ALTER USER roman IDENTIFIED BY kora;
```

Za pomocą polecenia PASSWORD można również zmienić hasło użytkownika, na konto którego jest się zalogowanym. Po wpisaniu PASSWORD SQL*Plus zażąda wpisania starego hasła oraz dwukrotnego wpisania nowego hasła (w celu potwierdzenia go). W poniższym przykładzie następuje połączenie z konta **roman** i uruchomienie PASSWORD (hasło jest oznaczone gwiazdkami):

```
CONNECT roman/kora
PASSWORD
Changing password for ROMAN
Old password:
New password:
Retype new password:
Password changed
```

Usuwanie konta użytkownika

Do usuwania kont użytkowników służy instrukcja DROP USER. W poniższym przykładzie następuje połączenie z konta użytkownika **system** i usunięcie konta użytkownika **roman**:

```
CONNECT system/oracle
DROP USER roman;
```



Uwaga Jeżeli schemat konta użytkownika zawiera obiekty, na przykład tabele, w instrukcji DROP USER należy umieścić za nazwą użytkownika słowo kluczowe CASCADE. Wcześniej należy się upewnić, czy żaden inny użytkownik nie potrzebuje dostępu do tych obiektów.

Uprawnienia systemowe

Uprawnienie systemowe umożliwia użytkownikowi wykonywanie określonych czynności w bazie danych, takich jak uruchamianie instrukcji DDL. Na przykład uprawnienie CREATE TABLE umożliwia użytkownikowi utworzenie tabeli w jego schemacie. Tabela 10.1 zawiera kilka często używanych uprawnień systemowych.



Uwaga Pełna lista uprawnień systemowych znajduje się w podręczniku *Oracle Database SQL Reference* opublikowanym przez Oracle Corporation.

Uprawnienia mogą być grupowane w **role**. Rolami, które zwykle warto przyznać użytkownikowi, są CONNECT i RESOURCE. CONNECT umożliwia nawiązanie połączenia z bazą danych, a RESOURCE — tworzenie różnych obiektów bazy danych, takich jak tabele, sekwencje kod PL/SQL itd.

Przyznawanie uprawnień systemowych użytkownikowi

Do przyznawania uprawnień systemowych użytkownikowi służy instrukcja GRANT. W poniższym przykładzie przyznano kilka uprawnień systemowych użytkownikowi **stefan**:

```
GRANT CREATE SESSION, CREATE USER, CREATE TABLE TO stefan;
```

Tabela 10.1. Często używane uprawnienia systemowe

Uprawnienie systemowe	Możliwości
CREATE SESSION	Nawiązanie połączenia z bazą danych
CREATE SEQUENCE	Utworzenie sekwencji będącej szeregiem liczb, używanym zwykle do automatycznego umieszczania danych w kolumnie klucza głównego. Sekwencje zostaną opisane w następnym rozdziale
CREATE SYNONYM	Utworzenie synonimu. Synonim umożliwia odwoływanie się do tabel w innym schemacie. Synonimy zostaną opisane w dalszej części tego rozdziału
CREATE TABLE	Utworzenie tabeli w schemacie użytkownika
CREATE ANY TABLE	Utworzenie tabeli w dowolnym schemacie
DROP TABLE	Usunięcie tabeli ze schematu użytkownika
DROP ANY TABLE	Usunięcie tabeli z dowolnego schematu
CREATE PROCEDURE	Utworzenie zachowanej procedury
EXECUTE ANY PROCEDURE	Uruchomienie procedury składowanej w dowolnym schemacie
CREATE USER	Utworzenie konta użytkownika
DROP USER	Usunięcie konta użytkownika
CREATE VIEW	Utworzenie perspektywy. Perspektywa jest składowanym zapytaniem umożliwiającym uzyskanie dostępu do wielu tabel i kolumn. Perspektywy można odpytywać w taki sam sposób jak tabele. Perspektywy zostaną opisane w następnym rozdziale

Można również użyć opcji WITH ADMIN OPTION, aby umożliwić danej osobie przyznanie uprawnienia innemu użytkownikowi. W poniższym przykładzie przyznano użytkownikowi **stefan** uprawnienie EXECUTE ANY PROCEDURE z opcją ADMIN:

```
GRANT EXECUTE ANY PROCEDURE TO stefan WITH ADMIN OPTION;
```

Dzięki temu z konta **stefan** można przyznać uprawnienie EXECUTE ANY PROCEDURE innemu użytkownikowi. W poniższym przykładzie łączymy się jako użytkownik **stefan** i przydzielamy uprawnienie EXECUTE ANY PROCEDURE użytkownikowi **gabi**:

```
CONNECT stefan/guzik
GRANT EXECUTE ANY PROCEDURE TO gabii;
```

Aby przyznać jakieś uprawnienie wszystkim użytkownikom, należy przyznać je obiekowi PUBLIC. W poniższym przykładzie łączymy się jako użytkownik **system** i przyznajemy PUBLIC uprawnienie EXECUTE ANY PROCEDURE:

```
CONNECT system/oracle
GRANT EXECUTE ANY PROCEDURE TO PUBLIC;
```

Od tej chwili każdy użytkownik bazy danych ma uprawnienie EXECUTE ANY PROCEDURE.

Sprawdzanie uprawnień systemowych przyznanych użytkownikowi

W celu przejrzenia uprawnień przyznanych użytkownikowi należy przesyłać zapytanie do tabeli **user_sys_privs**. Niektóre jej kolumny zostały opisane w tabeli 10.2.

Tabela 10.2. Wybrane kolumny z tabeli user_sys_privs

Kolumna	Typ	Opis
username	VARCHAR2(128)	Nazwa bieżącego konta użytkownika
privilege	VARCHAR2(40)	Przyznane uprawnienie systemowe
admin_option	VARCHAR2(3)	Informacja o tym, czy użytkownik może przyznać dane uprawnienie innemu użytkownikowi (YES lub NO)



Tabela `user_sys_privs` stanowi część słownika danych bazy danych Oracle. Są w nim składowane informacje o samej bazie danych.

W poniższym przykładzie łączymy się jako użytkownik `stefan` i wysyłamy zapytania do tabeli `user_sys_privs`:

```
CONNECT stefan/guzik
SELECT username, privilege, admin_option
FROM user_sys_privs
ORDER BY privilege;
```

USERNAME	PRIVILEGE	ADM
STEFAN	CREATE SESSION	NO
STEFAN	CREATE TABLE	NO
STEFAN	CREATE USER	NO
STEFAN	EXECUTE ANY PROCEDURE	YES
PUBLIC	EXECUTE ANY PROCEDURE	NO

W kolejnym przykładzie łączymy się jako użytkownik `gabi` i wysyłamy zapytania do tabeli `user_sys_privs`:

```
CONNECT gabi/tort
SELECT username, privilege, admin_option
FROM user_sys_privs
ORDER BY privilege;
```

USERNAME	PRIVILEGE	ADM
GABI	CREATE SESSION	NO
PUBLIC	EXECUTE ANY PROCEDURE	NO
GABI	EXECUTE ANY PROCEDURE	NO

Należy zauważyć, że użytkownik `gabi` ma uprawnienie `EXECUTE ANY PROCEDURE` przyznane wcześniej przez użytkownika `stefan`.

Zastosowanie uprawnień systemowych

Po przyznaniu użytkownikowi uprawnienia systemowego może on wykonywać określone zadania. Użytkownik `stefan` ma na przykład uprawnienie `CREATE USER`, może więc utworzyć konto użytkownika:

```
CONNECT stefan/guzik
CREATE USER robert IDENTIFIED BY lampa;
```

Jeżeli użytkownik `stefan` spróbuje użyć uprawnienia, którego mu nie przyznano, zostanie zwrócony błąd ORA-01031: `niewystarczające uprawnienia`. Na przykład użytkownik `stefan` nie ma uprawnienia `DROP USER`. Próba usunięcia konta użytkownika `robert` przez użytkownika `stefan` kończy się niepowodzeniem:

```
SQL> DROP USER robert;
DROP USER robert
*
ERROR at line 1:
ORA-01031: niewystarczające uprawnienia
```

Odbieranie uprawnień systemowych

Do odebrania użytkownikowi uprawnienia systemowego służy instrukcja `REVOKE`. W poniższym przykładzie łączymy się jako użytkownik `system` i odbieramy użytkownikowi `stefan` uprawnienie `CREATE TABLE`:

```
CONNECT system/oracle
REVOKE CREATE TABLE FROM stefan;
```

W kolejnym przykładzie użytkownikowi `stefan` jest odbierane uprawnienie `EXECUTE ANY PROCEDURE`:
`REVOKE EXECUTE ANY PROCEDURE FROM stefan;`

Gdy odbierzemy uprawnienie EXECUTE ANY PROCEDURE użytkownikowi `stefan`, który przypisał je wcześniej użytkownikowi `gabi`, użytkownik `gabi` wciąż będzie je miał:

```
CONNECT gabi/tort
SELECT username, privilege, admin_option
FROM user_sys_privs
ORDER BY username, privilege;
```

USERNAME	PRIVILEGE	ADM
GABI	CREATE SESSION	NO
GABI	EXECUTE ANY PROCEDURE	NO
PUBLIC	EXECUTE ANY PROCEDURE	NO

Uprawnienia obiektowe

Uprawnienie obiektowe pozwala użytkownikowi na wykonywanie określonych czynności na obiektach bazy danych, takich jak uruchamianie instrukcji DML na tabelach. Na przykład uprawnienie `INSERT ON store.products` pozwala użytkownikowi wstawiać wiersze do tabeli `products` w schemacie `store`. Tabela 10.3 zawiera kilka często wykorzystywanych uprawnień obiektowych.

Tabela 10.3. Często używane uprawnienia obiektowe

Uprawnienie obiektowe	Zezwala na...
SELECT	Wykonanie instrukcji SELECT
INSERT	Wykonanie instrukcji INSERT
UPDATE	Wykonanie instrukcji UPDATE
DELETE	Wykonanie instrukcji DELETE
EXECUTE	Uruchomienie składowanej procedury



Pełna lista uprawnień obiektowych jest dostępna w podręczniku *Oracle Database SQL Reference* opublikowanym przez Oracle Corporation.

Przyznawanie użytkownikowi uprawnień obiektowych

Do przyznawania uprawnień obiektowych służy instrukcja `GRANT`. W poniższym przykładzie łączymy się jako użytkownik `store` i przyznajemy użytkownikowi `stefan` uprawnienia obiektowe `SELECT`, `INSERT` i `UPDATE` do tabeli `products` oraz uprawnienie `SELECT` do tabeli `employees`:

```
CONNECT store/store_password
GRANT SELECT, INSERT, UPDATE ON store.products TO stefan;
GRANT SELECT ON store.employees TO stefan;
```

W kolejnym przykładzie przyznajemy użytkownikowi `stefan` uprawnienie `UPDATE` dla kolumn `last_name` i `salary`:

```
GRANT UPDATE (last_name, salary) ON store.employees TO stefan;
```

Możemy również użyć opcji `GRANT`, aby umożliwić użytkownikowi przyznawanie uprawnień innemu użytkownikowi. W poniższym przykładzie przyznajemy użytkownikowi `stefan` uprawnienie `SELECT` do tabeli `customers` z opcją `GRANT`:

```
GRANT SELECT ON store.customers TO stefan WITH GRANT OPTION;
```



Opcji `GRANT` używamy, aby umożliwić danej osobie przyznawanie innemu użytkownikowi uprawnienia obiektowego, a opcji `ADMIN`, aby umożliwić przyznawanie uprawnienia systemowego.

Dzięki temu użytkownik `stefan` może przyznać uprawnienie `SELECT ON store.customers` innej osobie. W poniższym przykładzie łączymy się jako `stefan` i przyznajemy to uprawnienie gabi:

```
CONNECT stefan/guzik
GRANT SELECT ON store.customers TO gabi;
```

Sprawdzanie przekazanych uprawnień

Wysyłając zapytania do tabeli `user_tab_privs_made`, możemy sprawdzić, jakie uprawnienia zostały przekazane innym przez danego użytkownika. W tabeli 10.4 opisano kolumny tej tabeli.

Tabela 10.4. Wybrane kolumny tabeli `user_tab_privs_made`

Kolumna	Typ	Opis
grantee	VARCHAR2(128)	Użytkownik, któremu przyznano uprawnienie
table_name	VARCHAR2(128)	Nazwa obiektu (takiego jak tabela), do którego zostało przyznanie uprawnienie
grantor	VARCHAR2(128)	Użytkownik, który przyznał uprawnienie
privilege	VARCHAR2(40)	Uprawnienie do obiektu
grantable	VARCHAR2(3)	Określa, czy użytkownik, któremu przyznano uprawnienie, może przydzielić je innej osobie (wartość YES lub NO)
hierarchy	VARCHAR2(3)	Określa, czy uprawnienie jest częścią hierarchii (wartość YES lub NO)

W poniższym przykładzie łączymy się jako użytkownik `store` i odpytujemy tabelę `user_tab_privs_made`. Ponieważ zawiera ona wiele wierszy, ograniczono je do tych, w których `table_name` ma wartość `PRODUCTS`:

```
CONNECT store/store_password
SELECT grantee, table_name, grantor, privilege, grantable, hierarchy
FROM user_tab_privs_made
WHERE table_name = 'PRODUCTS';
```

GRANTEE	TABLE_NAME	HIERARCHY	
GRANTOR	PRIVILEGE	GRANTABLE	HIERARCHY
STEFAN	PRODUCTS		
STORE	INSERT	NO	NO
STEFAN	PRODUCTS		
STORE	SELECT	NO	NO
STEFAN	PRODUCTS		
STORE	UPDATE	NO	NO

Wysyłając zapytania do tabeli `user_col_privs_made`, możemy sprawdzić, jakie uprawnienia obiekto-wie do kolumn przekazał użytkownik. Tabela 10.5 zawiera opis kolumn w tabeli `user_col_privs_made`.

Tabela 10.5. Wybrane kolumny tabeli `user_col_privs_made`

Kolumna	Typ	Opis
grantee	VARCHAR2(128)	Użytkownik, któremu przyznano uprawnienie
table_name	VARCHAR2(128)	Nazwa tabeli, do której zostało przyznanie uprawnienie
column_name	VARCHAR2(128)	Nazwa kolumny, do której zostało przyznanie uprawnienie
grantor	VARCHAR2(128)	Użytkownik, który przyznał uprawnienie
privilege	VARCHAR2(40)	Uprawnienie do obiektu
grantable	VARCHAR2(3)	Określa, czy użytkownik, któremu przyznano uprawnienie, może przydzielić je innej osobie (wartość YES lub NO)

W poniższym przykładzie zapytania są wysyłane do tabeli `user_col_privs_made`:

```
SELECT grantee, table_name, column_name, grantor, privilege, grantable
FROM user_col_privs_made
ORDER BY column_name;
```

GRANTEE	TABLE_NAME
COLUMN_NAME	GRANTOR
PRIVILEGE	GRA
STEFAN	EMPLOYEES
LAST_NAME	STORE
UPDATE	NO
STEFAN	EMPLOYEES
SALARY	STORE
UPDATE	NO

Sprawdzanie otrzymanych uprawnień obiektowych

Wysyłając zapytania do tabeli `user_tab_privs_recd`, możemy sprawdzić, jakie uprawnienia do tabel otrzymał dany użytkownik. W tabeli 10.6 opisano kolumny tej tabeli.

Tabela 10.6. Wybrane kolumny tabeli `user_tab_privs_recd`

Kolumna	Typ	Opis
owner	VARCHAR2(128)	Użytkownik będący właścicielem obiektu
table_name	VARCHAR2(128)	Nazwa obiektu, do którego zostało przyznanne uprawnienie
grantor	VARCHAR2(128)	Użytkownik, który przyznał uprawnienie
privilege	VARCHAR2(40)	Uprawnienie do obiektu
grantable	VARCHAR2(3)	Określa, czy użytkownik, któremu przyznano uprawnienie, może przydzielić je innej osobie (wartość YES lub NO)
hierarchy	VARCHAR2(3)	Określa, czy uprawnienie jest częścią hierarchii (wartość YES lub NO)

W poniższym przykładzie łączymy się jako użytkownik `stefan` i wysyłamy zapytania do tabeli `user_tab_privs_recd`:

```
CONNECT stefan/guzik
SELECT owner, table_name, grantor, privilege, grantable, hierarchy
FROM user_tab_privs_recd
ORDER BY table_name, privilege;
```

OWNER	TABLE_NAME	PRIVILEGE	GRA	HIE
STORE	CUSTOMERS			
STORE	SELECT		YES	NO
STORE	EMPLOYEES			
STORE	SELECT		NO	NO
STORE	PRODUCTS			
STORE	INSERT		NO	NO
STORE	PRODUCTS			
STORE	SELECT		NO	NO
STORE	PRODUCTS			
STORE	UPDATE		NO	NO

Wysyłając zapytania do tabeli `user_col_privs_recd`, możemy sprawdzić, jakie uprawnienia obiektowe do kolumn otrzymał użytkownik. Tabela 10.7 zawiera opis kolumn tej tabeli.

Tabela 10.7. Kolumny tabeli `user_col_privs_recd`

Kolumna	Typ	Opis
owner	VARCHAR2(128)	Użytkownik będący właścicielem obiektu
table_name	VARCHAR2(128)	Nazwa tabeli, do której zostało przyznanie uprawnienie
column_name	VARCHAR2(128)	Nazwa kolumny, do której zostało przyznanie uprawnienie
grantor	VARCHAR2(128)	Użytkownik, który przyznał uprawnienie
privilege	VARCHAR2(40)	Uprawnienie do obiektu
grantable	VARCHAR2(3)	Określa, czy użytkownik, któremu przyznano uprawnienie, może przydzielić je innej osobie (wartość YES lub NO)

W poniższym przykładzie wysyłamy zapytania do tabeli `user_col_privs_recd`:

```
SELECT owner, table_name, column_name, grantor, privilege, grantable
FROM user_col_privs_recd;
```

OWNER	TABLE_NAME
COLUMN_NAME	GRANTOR
PRIVILEGE	GRA
STORE	EMPLOYEES
LAST_NAME	STORE
UPDATE	NO
STORE	EMPLOYEES
SALARY	STORE
UPDATE	NO

Zastosowanie uprawnień obiektowych

Użytkownik może skorzystać z przyznanego mu uprawnienia, aby wykonać określone zadanie. Na przykład użytkownik `stefan` ma uprawnienie `SELECT` do `store.customers`:

```
CONNECT stefan/guzik
SELECT *
FROM store.customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Braz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Jeżeli użytkownik `stefan` spróbuje pobrać wiersze z tabeli `purchases`, do której nie ma żadnych uprawnień, zostanie zwrócony błąd ORA-00942: tabela lub perspektywa nie istnieje:

```
SQL> SELECT *
  2  FROM store.purchases;
FROM store.purchases
*
BŁĄD w linii 2:
ORA-00942: tabela lub perspektywa nie istnieje
```

Synonimy

Z przykładów zamieszczonych w poprzednim podrozdziale wynika, że możemy uzyskać dostęp do tabel w innym schemacie, poprzedzając ich nazwę nazwą schematu. Na przykład gdy użytkownik **stefan** pobierał wiersze z tabeli **customers** w schemacie **store**, wykonał zapytanie **store.customers**. Tworząc **synonim** dla tabeli, możemy uniknąć konieczności wprowadzania nazwy schematu. Służy do tego instrukcja **CREATE SYNONYM**.

Przejdźmy do przykładu. Zaczniemy od połączenia się z bazą danych jako użytkownik **system** i przyznaniu uprawnień **CREATE SYNONYM** użytkownikowi **stefan**:

```
CONNECT system/manager
GRANT CREATE SYNONYM TO stefan;
```

Następnie połączymy się jako użytkownik **stefan** i wykonamy instrukcję **CREATE SYNONYM** w celu utworzenia synonimu dla tabeli **store.customers**:

```
CONNECT stefan/guzik
CREATE SYNONYM customers FOR store.customers;
```

W celu pobrania wierszy z tabeli **store.customers** użytkownik **stefan** musi jedynie w klauzuli **FROM** instrukcji **INSERT** odwołać się do synonimu **customers**:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212
3	Stefan	Brąz	71/03/16	800-555-1213
4	Grażyna	Cynk		800-555-1214
5	Jadwiga	Mosiądz	70/05/20	

Synonimy publiczne

Dla tabeli można również utworzyć synonim **publiczny**. Wówczas będzie on widoczny dla wszystkich użytkowników. Poniższe instrukcje wykonują następujące zadania:

- połączenie z bazą danych jako użytkownik **system**,
- przyznanie uprawnień systemowego **CREATE PUBLIC SYNONYM** użytkownikowi **store**,
- połączenie z bazą danych jako użytkownik **store**,
- utworzenie dla tabeli **store.products** publicznego synonimu o nazwie **products**:

```
CONNECT system/manager
GRANT CREATE PUBLIC SYNONYM TO store;
CONNECT store/store_password
CREATE PUBLIC SYNONYM products FOR store.products;
```

Jeżeli połączymy się jako użytkownik **stefan**, który ma uprawnienie **SELECT** do tabeli **store.products**, będziemy mogli pobierać wiersze z tej tabeli za pomocą publicznego synonimu **products**:

```
CONNECT stefan/guzik
SELECT *
FROM products;
```

Mimo że utworzono publiczny synonim dla **store.products**, do uzyskania dostępu do tej tabeli użytkownik wciąż potrzebuje odpowiedniego uprawnienia obiektowego. Na przykład użytkownik **gabi** widzi publiczny synonim **products**, nie ma jednak uprawnień obiektowych do tabeli **store.products**. Dlatego jeśli spróbuje pobrać wiersze z **products**, zostanie zwrócony błąd ORA-00942: tabela lub perspektywa nie istnieje:

```
SQL> CONNECT gabi/tort
Connected.
```

```
SQL> SELECT * FROM products;
SELECT * FROM products
*
ERROR at line 1:
ORA-00942: tabela lub perspektywa nie istnieje
```

Gdyby użytkownik **gabi** miał uprawnienie obiektowe **SELECT** do tabeli **store.products**, przedstawione powyżej zapytanie powiodłoby się.

Jeżeli użytkownik ma inne uprawnienia obiektowe, może je wykorzystać za pośrednictwem synonimu. Na przykład gdyby użytkownik **gabi** miał uprawnienie obiektowe **INSERT** do tabeli **store.products**, mógłby wstawić wiersz do tej tabeli za pośrednictwem synonimu **products**.

Odbieranie uprawnień obiektowych

Do odbierania uprawnień obiektowych służy instrukcja **REVOKE**. W poniższym przykładzie łączymy się jako użytkownik **store** i odbieramy użytkownikowi **stefan** uprawnienie **INSERT** do tabeli **products**:

```
CONNECT store/store_password
REVOKE INSERT ON products FROM stefan;
```

W kolejnym przykładzie odbieramy użytkownikowi **stefan** uprawnienie **SELECT** do tabeli **customers**:

```
REVOKE SELECT ON customers FROM stefan;
```

Jeżeli odbierzemy uprawnienie **SELECT ON store.customers** użytkownikowi **stefan**, który przekazał je już użytkownikowi **gabi**, **gabi** również utraci to uprawnienie.

Role

Rola jest zbiorem uprawnień, które możemy przyznać użytkownikowi lub innej roli. Poniżej przedstawiono listę zalet i właściwości ról:

- zamiast przyznawać uprawnienia kolejno każdemu użytkownikowi, możemy utworzyć rolę, przyznać jej uprawnienia, a następnie przyznać tę rolę wielu użytkownikom i rolom,
- jeżeli dodamy lub usuniemy uprawnienie z roli, wszyscy użytkownicy i role, którym przyznano tę rolę, automatycznie otrzymają lub utracą to uprawnienie,
- użytkownikowi i roli możemy przydzielić kilka ról,
- roli można przyznać hasło.

Role są zatem przydatne w zarządzaniu uprawnieniami przypisanymi wielu użytkownikom.

Tworzenie ról

Do utworzenia roli niezbędne jest posiadanie uprawnienia **CREATE ROLE**. W jednym z kolejnych przykładów użytkownik **store** będzie również potrzebował możliwości przyznawania uprawnienia **CREATE USER** z opcją **ADMIN**. W poniższym przykładzie łączymy się jako użytkownik **system** i przyznajemy użytkownikowi **store** wszystkie niezbędne uprawnienia:

```
CONNECT system/oracle
GRANT CREATE ROLE TO store;
GRANT CREATE USER TO store WITH ADMIN OPTION;
```

Role tworzy się za pomocą instrukcji **CREATE ROLE**. W tabeli 10.8 przedstawiono role, które wkrótce utworzymy.

Do tworzenia ról służy instrukcja **CREATE ROLE**. Poniżej łączymy się jako użytkownik **store** i tworzymy role wymienione w tabeli 10.8:

```
CONNECT store/store_password
CREATE ROLE product_manager;
CREATE ROLE hr_manager;
CREATE ROLE overall_manager IDENTIFIED BY manager_password;
```

Tabela 10.8. Role do utworzenia

Nazwa roli	Uprawnienia
product_manager	Wykonywanie instrukcji SELECT, INSERT, UPDATE i DELETE w tabelach product_types i products
hr_manager	Wykonywanie instrukcji SELECT, INSERT, UPDATE i DELETE w tabelach salary_grades i employees, a także tworzenie kont użytkowników
overall_manager	Wykonywanie instrukcji SELECT, INSERT, UPDATE i DELETE w tabelach wymienionych powyżej. Roli overall_manager przyznano powyższe role

Roli overall_manager przypisano hasło manager_password.

Przyznawanie uprawnień roli

Do przyznawania uprawnień roli służy instrukcja GRANT. Roli możemy przyznać zarówno uprawnienia systemowe, jak i obiektywe, a także przyznać ją innej roli. W poniższym przykładzie przyznajemy niezbędne uprawnienia rojom product_manager i hr_manager, a także przyznajemy je obie roli overall_manager:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON product_types TO product_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON products TO product_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON salary_grades TO hr_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO hr_manager;
GRANT CREATE USER TO hr_manager;
GRANT product_manager, hr_manager TO overall_manager;
```

Przyznawanie roli użytkownikowi

Poniższe instrukcje tworzą użytkowników wykorzystywanych w tym rozdziale:

```
CONNECT system/oracle
CREATE USER stefan IDENTIFIED BY guzik;
CREATE USER henryk IDENTIFIED BY niebieski;
GRANT CREATE SESSION TO stefan;
GRANT CREATE SESSION TO henryk;
```

Do przyznawania roli użytkownikowi służy instrukcja GRANT. W poniższym przykładzie przyznajemy rolę hr_manager użytkownikowi henryk:

```
GRANT hr_manager TO henryk;
```

W następnym przykładzie przyznajemy rolę overall_manager użytkownikowi stefan:

```
GRANT overall_manager TO stefan;
```

Sprawdzanie ról przyznanych użytkownikowi

Wysyłając zapytania do tabeli user_role_privs, możemy sprawdzić, jakie role zostały przyznane użytkownikowi. Jej kolumny zostały opisane w tabeli 10.9.

Tabela 10.9. Kolumny tabeli user_role_privs

Kolumna	Typ	Opis
username	VARCHAR2(128)	Nazwa użytkownika, któremu przyznano rolę
granted_role	VARCHAR2(128)	Nazwa roli przyznanej użytkownikowi
admin_option	VARCHAR2(3)	Określa, czy użytkownik może przyznać tę rolę innej osobie (wartość YES lub NO)
default_role	VARCHAR2(3)	Określa, czy rola jest aktywowana domyślnie, gdy użytkownik łączy się z bazą danych (wartość YES lub NO)
os_granted	VARCHAR2(3)	Określa, czy rola została przyznana przez system operacyjny (wartość YES lub NO)

W poniższym przykładzie łączymy się jako użytkownik stefan i wysyłamy zapytania do tabeli user_role_privs:

```
CONNECT stefan/guzik
SELECT username, granted_role, admin_option, default_role, os_granted
FROM user_role_privs;
```

USERNAME	GRANTED_ROLE	ADM DEF OS_
STEFAN	OVERALL_MANAGER	NO YES NO

W kolejnym przykładzie łączymy się jako użytkownik `henryk` i wysyłamy zapytania do tabeli `user_role_privs`:

```
CONNECT henryk/niebieski
SELECT username, granted_role, admin_option, default_role, os_granted
FROM user_role_privs;
```

USERNAME	GRANTED_ROLE	ADM DEF OS_
HENRYK	HR_MANAGER	NO YES NO

Można zauważyć, że role zabezpieczone hasłem są nieaktywne. Kolumna `default_role` w powyższym przykładzie pokazuje, że rola `overall_manager` jest wyłączona. Ta rola jest zabezpieczona hasłem. Przed jej użyciem użytkownik musi wpisać hasło roli.

Rola jest domyślnie przyznawana jej twórcy. W poniższym przykładzie łączymy się jako użytkownik `store` i wysyłamy zapytania do tabeli `user_role_privs`:

```
CONNECT store/store_password
SELECT username, granted_role, admin_option, default_role, os_granted
FROM user_role_privs
ORDER BY username, granted_role;
```

USERNAME	GRANTED_ROLE	ADM DEF OS_
STORE	CONNECT	NO YES NO
STORE	HR_MANAGER	YES YES NO
STORE	OVERALL_MANAGER	YES NO NO
STORE	PRODUCT_MANAGER	YES YES NO
STORE	RESOURCE	NO YES NO

Należy zauważyć, że oprócz ról utworzonych przez siebie użytkownik `store` ma przyznane również role `CONNECT` i `RESOURCE`.



CONNECT i RESOURCE są rolami wbudowanymi, które zostały przyznane użytkownikowi `store` po uruchomieniu skryptu `store_schema.sql`. Jak przekonamy się w następnym rozdziale, role CONNECT i RESOURCE zawierają wiele uprawnień.

Sprawdzanie uprawnień systemowych przyznanych roli

Wysyłając zapytania do tabeli `role_sys_privs`, możemy sprawdzić, jakie uprawnienia systemowe zostały przyznane roli. Tabela 10.10 zawiera opis kolumn tej tabeli `role_sys_privs`.

Tabela 10.10. Wybrane kolumny tabeli `role_sys_privs`

Kolumna	Typ	Opis
<code>role</code>	VARCHAR2(128)	Nazwa roli
<code>privilege</code>	VARCHAR2(40)	Uprawnienie systemowe przyznane roli
<code>admin_option</code>	VARCHAR2(3)	Określa, czy uprawnienie zostało przyznane z opcją ADMIN (wartość YES lub NO)

W poniższym przykładzie pobieramy wiersze z tabeli `role_sys_privs` (zakładając, że wciąż jesteśmy połączeni jako użytkownik `store`):

```
SELECT role, privilege, admin_option
FROM role_sys_privs
ORDER BY privilege;
```

ROLE	PRIVILEGE	ADM
RESOURCE	CREATE CLUSTER	NO
RESOURCE	CREATE INDEXTYPE	NO
RESOURCE	CREATE OPERATOR	NO
RESOURCE	CREATE PROCEDURE	NO
RESOURCE	CREATE SEQUENCE	NO
CONNECT	CREATE SESSION	NO
RESOURCE	CREATE TABLE	NO
RESOURCE	CREATE TRIGGER	NO
RESOURCE	CREATE TYPE	NO
HR_MANAGER	CREATE USER	NO
CONNECT	SET CONTAINER	NO

Należy zauważyć, że rola RESOURCE ma przyznanych wiele uprawnień.



Powyższe zapytanie zostało uruchomione w Oracle Database 12c. Jeżeli jest wykorzystywana inna wersja oprogramowania bazy danych, wyniki mogą się nieco różnić od przedstawionych powyżej.

Sprawdzanie uprawnień obiektowych przyznanych roli

Wysyłając zapytania do tabeli `role_tab_privs`, możemy sprawdzić, jakie uprawnienia zostały przyznane roli. W tabeli 10.11 opisano kolumny tabeli `role_tab_privs`.

Tabela 10.11. Wybrane kolumny tabeli `role_tab_privs`

Kolumna	Typ	Opis
role	VARCHAR2(128)	Nazwa roli
owner	VARCHAR2(128)	Użytkownik będący właścicielem obiektu
table_name	VARCHAR2(128)	Nazwa obiektu, do którego zostało przyznane uprawnienie
column_name	VARCHAR2(128)	Nazwa kolumny (jeżeli dotyczy)
privilege	VARCHAR2(40)	Uprawnienie do obiektu
grantable	VARCHAR2(3)	Określa, czy uprawnienie zostało przyznane z opcją GRANT (wartość YES lub NO)

W poniższym przykładzie z tabeli `role_tab_privs` wybierane są wiersze, w których `role` ma wartość `HR_MANAGER`:

```
SELECT role, owner, table_name, column_name, privilege, grantable
FROM role_tab_privs
WHERE role='HR_MANAGER'
ORDER BY table_name, column_name, privilege;
```

ROLE	OWNER
TABLE_NAME	COLUMN_NAME
PRIVILEGE	GRA
HR_MANAGER	STORE
EMPLOYEES	
DELETE	NO
HR_MANAGER	STORE
EMPLOYEES	
INSERT	NO
HR_MANAGER	STORE
EMPLOYEES	
SELECT	NO

HR_MANAGER	STORE
EMPLOYEES	
UPDATE	NO
HR_MANAGER	STORE
SALARY_GRADES	
DELETE	NO
HR_MANAGER	STORE
SALARY_GRADES	
INSERT	NO
HR_MANAGER	STORE
SALARY_GRADES	
SELECT	NO
HR_MANAGER	STORE
SALARY_GRADES	
UPDATE	NO

Zastosowanie uprawnień przyznanych roli

Po otrzymaniu uprawnienia za pośrednictwem roli użytkownik może dzięki niemu wykonywać określone zadania. Na przykład użytkownik `henryk` ma przyznaną rolę `hr_manager`. Tej roli zostały przyznane uprawnienia `SELECT`, `INSERT`, `UPDATE` i `DELETE` dla tabel `salary_grades` i `employees`. Użytkownik `henryk` może zatem pobrać wiersze z tych tabel, co obrazuje poniższy przykład:

```
CONNECT henryk/niebieski
SELECT employee_id, last_name
FROM store.employees
WHERE salary < ANY
  (SELECT low_salary
   From store.salary_grades)
ORDER BY employee_id;
```

EMPLOYEE_ID	LAST_NAME
-----	-----
2	Józwierz
3	Helc
4	Nowak

W przypadku roli zabezpieczonej hasłem użytkownik musi wpisać hasło roli, zanim zacznie z roli korzystać. Na przykład użytkownik `stefan` ma zabezpieczoną hasłem rolę `overall_manager`. Zanim będzie mógł użyć tej roli, musi ją aktywować i podać hasło za pomocą instrukcji `SET ROLE`:

```
CONNECT stefan/guzik
SET ROLE overall_manager IDENTIFIED BY manager_password;
```

Wtedy `stefan` może korzystać z uprawnień przypisanych do roli. Na przykład:

```
SELECT employee_id, last_name
FROM store.employees
WHERE salary < ANY
  (SELECT low_salary
   FROM store.salary_grades)
ORDER BY employee_id;
```

EMPLOYEE_ID	LAST_NAME
-----	-----
2	Józwierz
3	Helc
4	Nowak

Aktywacja i deaktywacja ról

Rolę można deaktywować. Robi się to, modyfikując rolę za pomocą instrukcji ALTER ROLE w taki sposób, by nie była rolą domyślną. W poniższym przykładzie ustanawiane jest połączenie użytkownika system i użytkownik henryk jest modyfikowany w taki sposób, że hr_manager przestaje być rolą domyślną:

```
CONNECT system/oracle
ALTER USER henryk DEFAULT ROLE ALL EXCEPT hr_manager;
```

Kolejny przykład ustanawia połączenie użytkownika henryk i pokazuje, że hr_manager nie jest już rolą domyślną:

```
CONNECT henryk/niebieski
SELECT username, granted_role, default_role
FROM user_role_privs;
```

USERNAME	GRANTED_ROLE	DEF
HENRYK	HR_MANAGER	NO

Kolejne przykłady pokazują, że stefan nie ma dostępu do tabel salary_grades i employees:

```
SQL> SELECT * FROM store.salary_grades;
SELECT * FROM store.salary_grades
*
ERROR at line 1:
ORA-00942: tabela lub perspektywa nie istnieje

SQL> SELECT * FROM store.employees;
SELECT * FROM store.employees
*
ERROR at line 1:
ORA-00942: tabela lub perspektywa nie istnieje
```

Następny przykład aktywuje rolę hr_manager za pomocą polecenia SET ROLE:

```
SET ROLE hr_manager;
```

Gdy rola zostanie ustawiona, uprawnienia przydzielone tej roli mogą być wykorzystane.

Poprzedni przykład nie ustawia roli hr_manager jako roli domyślnej, co oznacza, że jeśli stefan się wyloguje i zaloguje ponownie, rola będzie niedostępna. Kolejny przykład ustawia rolę hr_manager jako domyślną, co zostanie zapisane po wylogowaniu:

```
CONNECT system/oracle
ALTER USER henryk DEFAULT ROLE hr_manager;
```

Możemy również ustawić rolę NONE (czyli brak roli), na przykład:

```
CONNECT henryk/niebieski
SET ROLE NONE;
```

Poniższy przykład pozwala ustawić swoją rolę na „wszystkie role” oprócz hr_manager, używając poniższej instrukcji:

```
SET ROLE ALL EXCEPT hr_manager
```

Odbieranie roli

Do odbierania roli służy instrukcja REVOKE. W poniższym przykładzie łączymy się jako użytkownik store i odbieramy rolę overall_manager użytkownikowi stefan:

```
CONNECT store/store_password
REVOKE overall_manager FROM stefan;
```

Odbieranie uprawnień roli

Do odbierania uprawnień roli służy instrukcja REVOKE. W poniższym przykładzie łączymy się jako użytkownik store i odbieramy roli product_manager wszystkie uprawnienia do tabel products i product_types (musimy być połączeni jako użytkownik store):

```
CONNECT store/store_password
REVOKE ALL ON products FROM product_manager;
REVOKE ALL ON product_types FROM product_manager;
```

Usuwanie roli

Do usuwania ról służy instrukcja DROP ROLE. W poniższym przykładzie łączymy się jako użytkownik `store` i usuwamy role `overall_manager`, `product_manager` i `hr_manager` (jako użytkownik `store`):

```
CONNECT store/store_password
DROP ROLE overall_manager;
DROP ROLE product_manager;
DROP ROLE hr_manager;
```

Obserwacja

Oprogramowanie bazy danych Oracle umożliwia obserwowanie operacji w bazie danych. Niektóre z nich mogą być obserwowane na poziomie ogólnym, na przykład zakończona niepowodzeniem próba zalogowania się do bazy danych, inne mogą być natomiast obserwowane na poziomie szczegółowym, na przykład pobieranie wierszy z konkretnej tabeli. Zwykle za aktywowanie obserwacji i monitorowanie wyników w celu wykrycia nadużyć zabezpieczeń jest odpowiedzialny administrator bazy danych. W tym podroziale zostaną przedstawione proste przykłady obserwacji wykonywanej za pomocą instrukcji AUDIT.

Uprawnienia wymagane do przeprowadzania obserwacji

Możliwość użycia instrukcji AUDIT mają jedynie użytkownicy z poniższymi uprawnieniami:

- Do obserwacji operacji wysokiego poziomu jest konieczne uprawnienie AUDIT SYSTEM. Przykładem takiej operacji jest wydanie *jakiegokolwiek* instrukcji SELECT, niezależnie od tabeli, której dotyczy.
- Do śledzenia operacji na konkretnych obiektach bazy danych jest potrzebne uprawnienie AUDIT ANY lub obiekt bazy danych musi znajdować się w schemacie użytkownika przeprowadzającego obserwację. Przykładem takiej operacji jest wydanie instrukcji SELECT dla określonej tabeli.

W poniższym przykładzie łączymy się z bazą danych jako użytkownik `system` i przyznajemy uprawnienia AUDIT SYSTEM i AUDIT ANY użytkownikowi `store`:

```
CONNECT system/oracle
GRANT AUDIT SYSTEM TO store;
GRANT AUDIT ANY TO store;
```

Przykłady obserwacji

W poniższym przykładzie łączymy się z bazą danych jako użytkownik `store` i obserwujemy wydawanie instrukcji CREATE TABLE:

```
CONNECT store/store_password
AUDIT CREATE TABLE;
```

W wyniku instrukcji AUDIT będą obserwowane wszystkie instrukcje CREATE TABLE. Poniższa tworzy na przykład prostą tabelę testową:

```
CREATE TABLE test (
    id INTEGER
);
```

Zapis obserwowania informacji o koncie użytkownika, na którym jesteśmy aktualnie zalogowani, możemy przejrzeć za pomocą perspektywy `USER_AUDIT_TRAIL`. Poniższy przykład przedstawia zapis obserwacji wygenerowany przez powyższą instrukcję CREATE TABLE:

```
SELECT username, extended_timestamp
FROM user_audit_trail
```

```
USERNAME
-----
EXTENDED_TIMESTAMP
-----
STORE
08/10/11 17:40:38,921000 +02:00
```

Możemy również obserwować wydawanie instrukcji przez konkretnych użytkowników. Poniższa instrukcja powoduje rozpoczęcie obserwacji wszystkich instrukcji **SELECT** wydanych przez użytkownika **store**:

```
AUDIT SELECT TABLE BY store;
```

Kolejna instrukcja powoduje rozpoczęcie obserwacji wszystkich instrukcji **INSERT**, **UPDATE** i **DELETE** wydanych przez użytkowników **store** i **stefan**:

```
AUDIT INSERT TABLE, UPDATE TABLE, DELETE TABLE BY store, stefan;
```

Można również obserwować wydawanie instrukcji pracujących na określonym obiekcie bazy danych. Poniższa instrukcja powoduje rozpoczęcie obserwacji wszystkich instrukcji **SELECT** wydanych dla tabeli **products**:

```
AUDIT SELECT ON store.products;
```

Kolejna instrukcja powoduje rozpoczęcie obserwacji wszystkich instrukcji wydawanych dla tabeli **employees**:

```
AUDIT ALL ON store.employees;
```

Za pomocą opcji **WHenever SUCCESSFUL** i **WHenever NOT SUCCESSFUL** możemy określić, kiedy powinna być wykonywana obserwacja:

- **WHenever SUCCESSFUL** powoduje, że obserwacja będzie prowadzona, gdy wykonywanie instrukcji zakończy się powodzeniem,
- **WHenever NOT SUCCESSFUL** powoduje, że obserwacja będzie prowadzona, gdy wykonywanie instrukcji zakończy się niepowodzeniem.

Domyślnie obserwacja jest prowadzona zawsze, niezależnie od powodzenia instrukcji.

W poniższym przykładzie użyto opcji **WHenever NOT SUCCESSFUL**:

```
AUDIT UPDATE TABLE BY stefan WHenever NOT SUCCESSFUL;
```

```
AUDIT INSERT TABLE WHenever NOT SUCCESSFUL;
```

W kolejnym przykładzie użyto opcji **WHenever SUCCESSFUL** do obserwacji tworzenia i usuwania kont użytkowników:

```
AUDIT CREATE USER, DROP USER BY store WHenever SUCCESSFUL;
```

W poniższym przykładzie użyto opcji **WHenever SUCCESSFUL** do obserwacji tworzenia i usuwania kont użytkowników przez użytkownika **store**:

```
AUDIT CREATE USER, DROP USER BY store WHenever SUCCESSFUL;
```

Dostępne są również opcje **BY SESSION** i **BY ACCESS**:

- **BY SESSION** powoduje, że zapis obserwacji jest rejestrowany raz, gdy ten sam typ instrukcji zostanie wydany podczas tej samej sesji użytkownika. Sesja użytkownika rozpoczyna się, kiedy użytkownik zaloguje się do bazy danych, a kończy się wraz z jego wylogowaniem.
- **BY ACCESS** powoduje, że jeden zapis obserwacji jest rejestrowany za każdym razem, gdy zostanie wydany ten sam typ instrukcji, niezależnie od sesji użytkownika.

Poniższe przykłady obrazują użycie opcji **BY SESSION** i **BY ACCESS**:

```
AUDIT SELECT ON store.products BY SESSION;
```

```
AUDIT DELETE ON store.employees BY ACCESS;
```

```
AUDIT INSERT, UPDATE ON store.employees BY ACCESS;
```

Do wyłączenia obserwacji wykorzystuje się instrukcję **NOAUDIT**. Na przykład:

```
NOAUDIT SELECT ON store.products;
NOAUDIT DELETE ON store.employees;
NOAUDIT INSERT, UPDATE ON store.employees;
```

Można wykorzystać ALL PRIVILEGES, by operować na wszystkich uprawnieniach. Na przykład:

```
AUDIT ALL PRIVILEGES ON store.products BY SESSION;
NOAUDIT ALL PRIVILEGES ON store.employees;
NOAUDIT ALL PRIVILEGES;
```

W Oracle Database 11g Release 2 wprowadzone zostały następujące opcje:

- **ALL STATEMENTS**, która opisuje wszystkie instrukcje SQL. Można wykorzystać ALL STATEMENTS z połeceniami AUDIT i NOAUDIT.
- **IN SESSION CURRENT**, która ogranicza obserwację do bieżącej sesji bazy danych. Można wykorzystać IN SESSION CURRENT w instrukcji AUDIT.

Kolejny przykład pokazuje użycie opcji ALL STATEMENTS i IN SESSION CURRENT:

```
AUDIT ALL STATEMENTS;
AUDIT ALL STATEMENTS IN SESSION CURRENT BY ACCESS WHENEVER NOT SUCCESSFUL;
NOAUDIT ALL STATEMENTS;
```

Perspektywy zapisu obserwacji

Wcześniej zostało zaprezentowane zastosowanie perspektywy USER_AUDIT_TRAIL. Ta i inne perspektywy zapisu obserwacji zostały opisane poniżej:

- **USER_AUDIT_OBJECT** wyświetla zapis obserwacji dla wszystkich obiektów dostępnych dla bieżącego użytkownika,
- **USER_AUDIT_SESSION** wyświetla zapis obserwacji dla połączeń i rozłączeń bieżącego użytkownika,
- **USER_AUDIT_STATEMENT** wyświetla zapis obserwacji instrukcji GRANT, REVOKE, AUDIT, NOAUDIT i ALTER SYSTEM wydanych przez bieżącego użytkownika,
- **USER_AUDIT_TRAIL** wyświetla wszystkie zapisy obserwacji powiązane z bieżącym użytkownikiem.

Z pomocą tych perspektyw możemy zbadać zapisy obserwacji. W bazie danych jest wiele dostępnych perspektyw o podobnych nazwach, które umożliwiają administratorowi bazy danych przeglądanie zapisów obserwacji. Są to między innymi: DBA_AUDIT_OBJECT, DBA_AUDIT_SESSION, DBA_AUDIT_STATEMENT i DBA_AUDIT_TRAIL. Umożliwiają one administratorowi przeglądanie zapisów obserwacji dla wszystkich użytkowników. Szczegółowe informacje na temat tych perspektyw znajdują się w podręczniku *Oracle Database Reference* opublikowanym przez Oracle Corporation.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- do tworzenia kont użytkowników służy instrukcja CREATE USER,
- uprawnienia systemowe umożliwiają wykonywanie określonych czynności w bazie danych, na przykład uruchamianie instrukcji DDL,
- uprawnienia obiektowe umożliwiają wykonywanie określonych czynności na obiektach bazy danych, na przykład uruchamianie instrukcji DML operujących na tabelach,
- tworząc synonim dla tabeli, możemy uniknąć wpisywania nazwy schematu,
- rola jest grupą uprawnień, które możemy przyznać użytkownikowi lub innej roli,
- do obserwacji wykonywania instrukcji SQL służy instrukcja AUDIT.

W kolejnym rozdziale zamieszczono informacje na temat tworzenia tabel, a także tego, jak tworzyć indeksy, sekwencje i perspektywy.

ROZDZIAŁ

11

Tworzenie tabel, sekwencji, indeksów i perspektyw

Z tego rozdziału dowiesz się, jak:

- tworzyć, modyfikować i usuwać tabele,
- tworzyć sekwencje generujące serie liczb i z nich korzystać,
- tworzyć indeksy mogące poprawić wydajność zapytań i używać ich,
- tworzyć perspektywy, które są wcześniej zdefiniowanymi zapytaniami i korzystać z nich,
- korzystać z danych zgromadzonych w archiwach migawek, w których są składowane zmiany wprowadzane w określonym przedziale czasu.

Tabele

Z tego podrozdziału dowiesz się więcej o tworzeniu tabel. Zobaczysz, jak zmieniać i usuwać tabele oraz w jaki sposób pobiera się informacje o tabeli ze słownika danych. Słownik danych zawiera informacje o wszystkich elementach bazy danych, takich jak tabele, sekwencje, indeksy itd.

Tworzenie tabeli

Do tworzenia tabel służy instrukcja `CREATE TABLE`. Jej uproszczona składnia ma następującą postać:

```
CREATE [GLOBAL TEMPORARY] TABLE nazwa_tabeli (  
    nazwa_kolumny typ [CONSTRAINT def_więzów DEFAULT wyr_domyślnie]  
    [, nazwa_kolumny typ [CONSTRAINT def_więzów DEFAULT wyr_domyślnie] ...]  
)  
[ON COMMIT {DELETE | PRESERVE} ROWS]  
TABLESPACE przestrzeń_tabel;
```

gdzie:

- `GLOBAL TEMPORARY` oznacza, że wiersze tabeli są tymczasowe (tego typu tabele nazywamy tymczasowymi). Wiersze w takiej tabeli są ważne w sesji użytkownika, a to, jak długo będą obowiązywać, jest określone klauzulą `ON COMMIT`.
- `nazwa_tabeli` jest nazwą tabeli.
- `nazwa_kolumny` jest nazwą kolumny.
- `typ` jest typem kolumny.
- `def_więzów` jest więzami integralności kolumny.
- `wyr_domyślnie` jest wyrażeniem przypisanym jako domyślna wartość kolumny.

- ON COMMIT określa czas składowania wierszy w tabeli tymczasowej. DELETE oznacza, że wiersze będą usuwane po zakończeniu transakcji. PRESERVE oznacza, że wiersze będą składowane do końca sesji użytkownika — wówczas zostaną usunięte. Jeżeli klauzula ON COMMIT zostanie pominięta w przypadku tabeli tymczasowej, zostanie użyte domyślne ustawienie — DELETE.
- przestrzeń_tabel jest przestrzenią tabel dla tabeli. Jeżeli nie zostanie określona, tabela będzie składowana w domyślnej przestrzeni tabel użytkownika.



Pelna składnia instrukcji CREATE TABLE jest znacznie bardziej rozbudowana. Szczegółowe informacje na ten temat są dostępne w książce *Oracle Database SQL Reference* opublikowanej przez Oracle Corporation.

W poniższym przykładzie łączymy się jako użytkownik store i tworzymy tabelę o nazwie order_status2:

```
CONNECT store/store_password
CREATE TABLE order_status2 (
    id INTEGER CONSTRAINT order_status2_pk PRIMARY KEY,
    status VARCHAR2(10),
    last_modified DATE DEFAULT SYSDATE
);
```



W celu wykonania przykładów należy samodzielnie wpisywać i uruchamiać prezentowane instrukcje SQL w programie SQL*Plus.

W kolejnym przykładzie tworzymy tymczasową tabelę o nazwie order_status_temp, której wiersze będą składowane do końca sesji użytkownika (ON COMMIT PRESERVE ROWS):

```
CREATE GLOBAL TEMPORARY TABLE order_status_temp (
    id INTEGER,
    status VARCHAR2(10),
    last_modified DATE DEFAULT SYSDATE
)
ON COMMIT PRESERVE ROWS;
```

W kolejnym przykładzie:

- wstawiamy wiersz do tabeli order_status_temp,
- odłączamy się od bazy danych w celu zakończenia sesji, co powoduje usunięcie wiersza z tabeli order_status_temp,
- ponownie łączymy się z bazą danych jako użytkownik store i wysyłamy zapytania do tabeli order_status_temp, aby przekonać się, że nie zawiera żadnych wierszy:

```
INSERT INTO order_status_temp (
    id, status
) VALUES (
    1, 'Nowy'
);
```

1 row created.

```
DISCONNECT
CONNECT store/store_password
SELECT *
FROM order_status_temp;
```

no rows selected

Pobieranie informacji o tabelach

Informacje o tabelach możemy uzyskać przez:

- uruchomienie polecenia DESCRIBE dla tabeli,
- wysyłanie zapytań do perspektywy user_tables, stanowiącej część słownika danych.

W tabeli 11.1 znajduje się opis niektórych kolumn perspektywy `user_tables`.

Tabela 11.1. Wybrane kolumny perspektywy `user_tables`

Kolumna	Typ	Opis
<code>table_name</code>	<code>VARCHAR2(128)</code>	Nazwa tabeli
<code>tablespace_name</code>	<code>VARCHAR2(30)</code>	Nazwa przestrzeni tabel, w której jest składowana tabela. Przestrzeń tabel jest obszarem wykorzystywanym przez bazę danych do składowania obiektów takich jak tabele
<code>temporary</code>	<code>VARCHAR2(1)</code>	Określa, czy tabela jest tymczasowa. Jeżeli tak, w kolumnie znajduje się wartość Y. Jeżeli nie, w kolumnie znajduje się wartość N

W poniższym przykładzie pobieramy niektóre kolumny z tabeli `user_tables`, w której `table_name` ma wartość `order_status2` lub `order_status_temp`:

```
SELECT table_name, tablespace_name, temporary
FROM user_tables
WHERE table_name IN ('ORDER_STATUS2', 'ORDER_STATUS_TEMP');
```

TABLE_NAME	TABLESPACE_NAME	T
ORDER_STATUS2	USERS	N
ORDER_STATUS_TEMP		Y

Należy zauważyć, że tabela `ORDER_STATUS_TEMP` jest tymczasowa, na co wskazuje wartość Y w ostatniej kolumnie.



Wysyłając zapytania do perspektywy `all_tables`, możemy uzyskać informacje na temat wszystkich tabel, do których mamy dostęp.

Uzyskiwanie informacji o kolumnach w tabeli

Informacje o kolumnach w tabelach możemy pobrać z perspektywy `user_tab_columns`. W tabeli 11.2 zostały opisane niektóre jej kolumny.

Tabela 11.2. Wybrane kolumny perspektywy `user_tab_columns`

Kolumna	Typ	Opis
<code>table_name</code>	<code>VARCHAR2(128)</code>	Nazwa tabeli
<code>column_name</code>	<code>VARCHAR2(128)</code>	Nazwa kolumny
<code>data_type</code>	<code>VARCHAR2(128)</code>	Typ danych w kolumnie
<code>data_length</code>	<code>NUMBER</code>	Długość danych
<code>data_precision</code>	<code>NUMBER</code>	Precyza kolumny liczbowej, jeżeli została określona
<code>data_scale</code>	<code>NUMBER</code>	Skala kolumny liczbowej

W poniższym przykładzie z perspektywy `user_tab_columns` są pobierane niektóre kolumny zawierające informacje o kolumnach w tabeli `products`:

```
COLUMN column_name FORMAT a15
COLUMN data_type FORMAT a10
SELECT column_name, data_type, data_length, data_precision, data_scale
FROM user_tab_columns
WHERE table_name = 'PRODUCTS';
```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE
PRODUCT_ID	NUMBER	22	0	
PRODUCT_TYPE_ID	NUMBER	22	0	
NAME	VARCHAR2	30		

DESCRIPTION	VARCHAR2	50			
PRICE	NUMBER	22	5	2	



Wysyłając zapytania do perspektywy `all_columns`, możemy uzyskać informacje na temat wszystkich kolumn, do których mamy dostęp.

Zmienianie tabeli

Do wprowadzania zmian w tabeli służy instrukcja `ALTER TABLE`. Za jej pomocą możemy na przykład wykonać poniższe zadania:

- dodać, zmienić lub usunąć kolumnę,
- dodać lub usunąć więzy integralności,
- łączyć lub wyłączyć więzy integralności.

Z kolejnych podrozdziałów dowiesz się, jak używać instrukcji `ALTER TABLE` do wykonywania tych zadań. Dowiesz się też, jak uzyskać informacje na temat więzów integralności.

Wstawianie kolumny

W poniższym przykładzie użyto instrukcji `ALTER TABLE` do wstawienia do tabeli `order_status2` kolumny `modified_by` typu `INTEGER`:

```
ALTER TABLE order_status2
ADD modified_by INTEGER;
```

W kolejnym przykładzie do tabeli `order_status2` jest wstawiana kolumna o nazwie `initially_created`:

```
ALTER TABLE order_status2
ADD initially_created DATE DEFAULT SYSDATE NOT NULL;
```

Uruchamiając polecenie `DESCRIBE` dla tabeli `order_status2`, możemy sprawdzić, czy kolumny zostały wstawione:

Nazwa	Wartość NULL? Typ
ID	NOT NULL NUMBER(38)
STATUS	VARCHAR2(10)
LAST_MODIFIED	DATE
MODIFIED_BY	NUMBER(38)
INITIALLY_CREATED	NOT NULL DATE

Wstawianie kolumny wirtualnej

W Oracle Database 11g wprowadzono kolumny wirtualne. Kolumna wirtualna to taka, która jedynie odwołuje się do innych kolumn znajdujących się w tabeli. Na przykład poniższa instrukcja `ALTER TABLE` wstawia do tabeli `salary_grades` wirtualną kolumnę o nazwie `average_salary`:

```
ALTER TABLE salary_grades
ADD (average_salary AS ((low_salary + high_salary)/2));
```

Wartość w kolumnie `average_salary` jest ustawiona jako średnia wartości w kolumnach `low_salary` i `high_salary`.

Poniższe polecenie `DESCRIBE` potwierdza wstawienie kolumny `average_salary` do tabeli `salary_grades`:

Nazwa	Wartość NULL? Typ
SALARY_GRADE_ID	NOT NULL NUMBER(38)
LOW_SALARY	NUMBER(6)
HIGH_SALARY	NUMBER(6)
AVERAGE_SALARY	NUMBER

Poniższe zapytanie pobiera wiersze z tabeli salary_grades:

```
SELECT *
FROM salary_grades;
```

SALARY_GRADE_ID	LOW_SALARY	HIGH_SALARY	AVERAGE_SALARY
1	1	250000	125000,5
2	250001	500000	375000,5
3	500001	750000	625000,5
4	750001	999999	875000

Modyfikowanie kolumny

Wybrane właściwości kolumn, które można zmienić za pomocą instrukcji ALTER TABLE, to między innymi:

- rozmiar kolumny (jeżeli typ danych na to pozwala, na przykład CHAR lub VARCHAR2),
- precyza kolumny liczbowej,
- typ danych w kolumnie,
- domyślana wartość kolumny.

W kolejnych podrozdziałach zostaną przedstawione przykłady zmieniania tych właściwości kolumn.

Zmienianie rozmiaru kolumny

Poniższa instrukcja ALTER TABLE zwiększa maksymalną długość kolumny order_status2.status do 15 znaków:

```
ALTER TABLE order_status2
MODIFY status VARCHAR2(15);
```

 Ostrzeżenie Długość kolumny może być zmniejszona, tylko jeśli tabela nie zawiera żadnych wierszy lub wszystkie wiersze zawierają wartość NULL w tej kolumnie.

Zmienianie precyzji kolumny liczbowej

Poniższa instrukcja ALTER TABLE zmienia precyzę kolumny id na 5:

```
ALTER TABLE order_status2
MODIFY id NUMBER(5);
```

 Ostrzeżenie Precyza kolumny liczbowej może być zmniejszona, tylko jeżeli tabela nie zawiera żadnych wierszy lub wszystkie wiersze zawierają wartość NULL w tej kolumnie.

Zmienianie typu danych w kolumnie

Poniższa instrukcja ALTER TABLE zmienia typ danych w kolumnie status na CHAR:

```
ALTER TABLE order_status2
MODIFY status CHAR(15);
```

Jeżeli tabela jest pusta lub kolumna zawiera same wartości NULL, można zmienić typ danych na dowolny (między innymi na krótszy). W przeciwnym razie możemy zmienić typ danych w kolumnie jedynie na kompatybilny. Możemy na przykład zmienić VARCHAR2 na CHAR (i odwrotnie), jeżeli tylko nie skróćmy przy okazji kolumny. Nie możemy zmienić typu DATE na NUMBER.

Zmienianie domyślnej wartości kolumny

Poniższa instrukcja ALTER TABLE zmienia domyślną wartość kolumny last_modified na SYSDATE - 1:

```
ALTER TABLE order_status2
MODIFY last_modified DEFAULT SYSDATE - 1;
```

Domyślna wartość będzie stosowana jedynie dla nowych wierszy wstawianych do tabeli. W nowych wierszach w kolumnie last_modified będzie wpisywana bieżąca data minus jeden dzień.

Usuwanie kolumny

Poniższa instrukcja ALTER TABLE usuwa kolumnę `initially_created`:

```
ALTER TABLE order_status2
DROP COLUMN initially_created;
```

Dodawanie więzów

We wcześniejszych rozdziałach prezentowano przykłady tabel z więzami PRIMARY KEY, FOREIGN KEY i NOT NULL. Te oraz inne więzy zostały opisane w tabeli 11.3.

Tabela 11.3. Więzy i ich znaczenie

Więzy	Typ więzów	Znaczenie
CHECK	C	Wartość w kolumnie lub grupie kolumn musi spełniać określony warunek
NOT NULL	C	W kolumnie nie mogą być składowane wartości NULL. Jest to odmiana więzów CHECK
PRIMARY KEY	P	Klucz główny tabeli, składający się z jednej lub kilku kolumn, które identyfikują każdy wiersz tabeli w sposób unikatowy
FOREIGN KEY	R	Klucz obcy tabeli, odwołujący się do kolumny w innej tabeli lub kolumny w tej samej tabeli (samoodwołanie)
UNIQUE	U	W kolumnie lub grupie kolumn mogą być składowane jedynie wartości unikatowe
CHECK OPTION	V	Zmiany w wierszach tabeli wprowadzane za pośrednictwem perspektywy muszą najpierw zostać sprawdzone (więcej informacji na ten temat znajduje się w podrozdziale „Perspektywy”)
READ ONLY	O	Perspektywa może być jedynie odczytywana (więcej informacji na ten temat znajduje się w podrozdziale „Perspektywy”)

W kolejnych podrozdziałach zostały zawarte informacje o tym, jak dodać do tabeli `order_status2` niektóre więzy opisane w tabeli 11.3.

Dodawanie więzów CHECK

Poniższa instrukcja ALTER TABLE dodaje więzy CHECK do tabeli `order_status2`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_ck
CHECK (status IN ('ZŁOZONE', 'OCZEKUJĄCE', 'WYSŁANE'));
```

Więzy te wymuszają, aby w kolumnie `status` zawsze znajdowała się wartość ZŁOZONE, OCZEKUJĄCE lub WYSŁANE.

Poniższa instrukcja INSERT wstawia wiersz do tabeli `order_status2` (do kolumny `status` jest wpisywana wartość OCZEKUJĄCE):

```
INSERT INTO order_status2 (
    id, status, last_modified, modified_by
) VALUES (
    1, 'OCZEKUJĄCE', '05/01/01', 1
);
```

Jeżeli spróbujemy wstawić wiersz, który nie spełnia więzów CHECK, zostanie zwrócony błąd ORA-02290. Na przykład poniższa instrukcja INSERT próbuje wstawić wiersz, w którym kolumna `status` ma wartość spoza listy:

```
INSERT INTO order_status2 (
    id, status, last_modified, modified_by
) VALUES (
    2, 'ZREALIZOWANE', '05/01/01', 2
);
```

```
INSERT INTO order_status2 (
*
ERROR at line 1:
ORA-02290: naruszono więzy CHECK (STORE.ORDER_STATUS2_STATUS_CK)
```

Ponieważ więzy CHECK zostały naruszone, wiersz nie zostanie wstawiony do tabeli.

W więzach CHECK mogą być stosowane operatory porównania. W kolejnym przykładzie dodano ograniczenie, wymuszające, aby wartość id była większa od zera:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_id_ck CHECK (id > 0);
```

Gdy dodajemy ograniczenie, istniejące wiersze muszą spełniać. Gdyby na przykład tabela order_status2 zawierała wiersze, wartości w ich kolumnach id miałyby być większe od zera.



Uwaga Od zasady, że istniejące wiersze muszą spełniać ograniczenia, są wyjątki: więzy mogą być wyłączane, gdy je dodajemy, i można je ustawić tak, aby były stosowane jedynie do nowych danych (za pomocą opcji ENABLE NOVALIDATE). Z dalszej części rozdziału dowiesz się więcej na ten temat.

Dodawanie więzów NOT NULL

Poniższa instrukcja ALTER TABLE dodaje więzy NOT NULL do kolumny status tabeli order_status2. Zwróć uwagę, że dodajesz więzy NOT NULL za pomocą klauzuli MODIFY:

```
ALTER TABLE order_status2
MODIFY status CONSTRAINT order_status2_status_nn NOT NULL;
```

W kolejnym przykładzie więzy NOT NULL są dodawane do kolumny modified_by:

```
ALTER TABLE order_status2
MODIFY modified_by CONSTRAINT order_status2_modified_by_nn NOT NULL;
```

Poniższy przykład dodaje więzy NOT NULL do kolumny last_modified:

```
ALTER TABLE order_status2
MODIFY last_modified NOT NULL;
```

Należy zauważać, że nie została podana nazwa więzów. W takim przypadku oprogramowanie bazy danych automatycznie przypisze więzom nieczytelną nazwę, taką jak SYS_C003381.



Wskazówka Należy zawsze nadawać więzom zrozumiałe nazwy. Dzięki temu, jeżeli wystąpi błąd, będzie można łatwo określić jego przyczynę.

Dodawanie więzów FOREIGN KEY

Zanim przejdziemy do przykładu dodawania więzów FOREIGN KEY, usuniemy kolumnę modified_by z tabeli order_status2 za pomocą poniżej instrukcji ALTER TABLE:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

Kolejna instrukcja dodaje więzy FOREIGN KEY, odwołujące się do kolumny employee_id z tabeli employees:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id);
```

Zastosowanie klauzuli ON DELETE CASCADE dla więzów FOREIGN KEY sprawia, że gdy w tabeli nadrzędnej zostanie usunięty wiersz, wszystkie związane z nim wiersze w tabeli podrzędnej również zostaną usunięte. Poniżej jest usuwana kolumna modified_by i poprzedni przykład jest zmieniany tak, aby zawierał klauzulę ON DELETE CASCADE:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

```
ALTER TABLE order_status2
```

```
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id) ON DELETE CASCADE;
```

Jeżeli z tabeli `employees` zostanie usunięty wiersz, wszystkie związane z nim wiersze w tabeli `order_status2` również zostaną usunięte.

Zastosowanie klauzuli `ON DELETE SET NULL` dla więzów FOREIGN KEY sprawia, że gdy w tabeli nadzędnej zostanie usunięty wiersz, w kolumnie klucza obcego wiersza (lub wierszy) w tabeli podrzędnej zostanie wstawiona wartość `NULL`. Poniżej jest usuwana kolumna `modified_by` i poprzedni przykład jest zmieniany tak, aby zawierał klauzulę `ON DELETE SET NULL`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id) ON DELETE SET NULL;
```

Jeżeli z tabeli `employees` zostanie usunięty wiersz, w kolumnie `modified_by` we wszystkich związanych z nim wierszach tabeli `order_status2` zostanie wpisana wartość `NULL`.

Zanim przejdziemy do następnego podrozdziału, uporządkujemy trochę kod — poniższa instrukcja usuwa kolumnę `modified_by`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

Dodawanie więzów UNIQUE

Poniższa instrukcja `ALTER TABLE` dodaje więzy `UNIQUE` do kolumny `status`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status);
```

Każdy istniejący lub wstawiany wiersz musi mieć unikatową wartość w kolumnie `status`.

Usuwanie więzów

Do usuwania więzów służy klauzula `DROP CONSTRAINT` instrukcji `ALTER TABLE`. W poniższym przykładzie są usuwane więzy `order_status2_status_uq`:

```
ALTER TABLE order_status2
DROP CONSTRAINT order_status2_status_uq;
```

Wyłączanie więzów

Domyślnie po utworzeniu więzy są włączone. Możemy je jednak początkowo wyłączyć, dodając opcję `DISABLE` na końcu klauzuli `CONSTRAINT`. Poniższy przykład dodaje więzy do tabeli `order_status2`, ale jednocześnie je wyłącza:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status) DISABLE;
```

Istniejące więzy możemy wyłączyć za pomocą klauzuli `DISABLE CONSTRAINT` instrukcji `ALTER TABLE`. W poniższym przykładzie wyłączone są więzy `order_status2_status_nn`:

```
ALTER TABLE order_status2
DISABLE CONSTRAINT order_status2_status_nn;
```

Jeżeli za klauzulą `DISABLE CONSTRAINT` umieścimy opcję `CASCADE`, wszystkie więzy zależne od więzów określonych w klauzuli również zostaną wyłączone. Opcja `CASCADE` jest używana przy wyłączaniu więzów klucza głównego lub unikatowości, będącego częścią więzów klucza obcego innej tabeli.

Włączanie więzów

Do włączania więzów służy klauzula `ENABLE CONSTRAINT` instrukcji `ALTER TABLE`. W poniższym przykładzie włączane są więzy `order_status2_status_uq`:

```
ALTER TABLE order_status2
ENABLE CONSTRAINT order_status2_status_uq;
```

Aby włączyć więzy, wszystkie wiersze tabeli muszą je spełniać. Jeżeli na przykład tabela `order_status2` zawierałaby wiersze, w kolumnie `status` musiałaby się znajdować tylko unikatowe wartości.

Jeżeli użyjemy opcji `ENABLE NOVALIDATE`, więzy będą stosowane jedynie do nowych danych, na przykład:

```
ALTER TABLE order_status2
ENABLE NOVALIDATE CONSTRAINT order_status2_status_uq;
```



Domyślnie jest włączona opcja `ENABLE VALIDATE`, co oznacza, że istniejące wiersze muszą spełniać kryteria więzów.

Więzy odroczone

Więzy odroczone to takie, które są wymuszane przy zatwierdzeniu transakcji. Klauzułę `DEFERRABLE` można zastosować jedynie przy tworzeniu więzów. Po dodaniu nie można ich zmienić na `DEFERRABLE` — wówczas trzeba je usunąć i utworzyć na nowo.

Dodając więzy odroczone, możemy je oznaczyć jako `INITIALLY IMMEDIATE` lub `INITIALLY DEFERRED`. `INITIALLY IMMEDIATE` oznacza, że więzy będą sprawdzane za każdym razem, gdy dodajemy, modyfikujemy lub usuwamy wiersze z tabeli (czyli jak przy domyślnym zachowaniu więzów). `INITIALLY DEFERRED` oznacza, że więzy będą sprawdzane jedynie przy zatwierdzeniu transakcji.

Poniższa instrukcja usuwa więzy `order_status2_status_uq`:

```
ALTER TABLE order_status2
DROP CONSTRAINT order_status2_status_uq;
```

Kolejny przykład dodaje więzy `order_status2_status_uq`, oznaczając je jako `DEFERRABLE INITIALLY DEFERRED`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status)
DEFERRABLE INITIALLY DEFERRED;
```

Jeżeli dodamy wiersze do tabeli `order_status2`, więzy `order_status2_status_uq` nie będą wymuszane do momentu wydania instrukcji `COMMIT`.

Pobieranie informacji o więzach

Wysyłając zapytania do perspektywy `user_constraints`, możemy uzyskać informacje o więzach. Tabela 11.4 zawiera opis niektórych kolumn tej perspektywy.

Tabela 11.4. Wybrane kolumny perspektywy `user_constraints`

Kolumna	Typ	Opis
owner	VARCHAR2(128)	Właściciel więzów
constraint_name	VARCHAR2(128)	Nazwa więzów
constraint_type	VARCHAR2(1)	Typ więzów (P, R, C, U, V lub O). Znaczenie typów więzów zostało opisane w tabeli 11.3
table_name	VARCHAR2(128)	Nazwa tabeli, dla której więzy są zdefiniowane
status	VARCHAR2(8)	Stan więzów (ENABLED lub DISABLED — włączone lub wyłączone)
deferrable	VARCHAR2(14)	Określa, czy więzy są odroczone (DEFERRABLE lub NOT DEFERRABLE)
deferred	VARCHAR2(9)	Określa, czy więzy są wymuszane natychmiast, czy też są odroczone (IMMEDIATE lub DEFERRED)

Poniższy przykład pobiera z perspektywy `user_constraints` niektóre kolumny z informacjami o tabeli `order_status2`:

```
SELECT constraint_name, constraint_type, status, deferrable, deferred
FROM user_constraints
```

```
WHERE table_name = 'ORDER_STATUS2'
ORDER BY constraint_name;
```

CONSTRAINT_NAME	C	STATUS	DEFERRABLE	DEFERRED
ORDER_STATUS2_ID_CK	C	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_PK	P	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_CK	C	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_NN	C	DISABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_UQ	U	ENABLED	DEFERRABLE	DEFERRED
SYS_C004807	C	ENABLED	NOT DEFERRABLE	IMMEDIATE

Należy zauważać, że poza jednym wszystkie więzy mają użyteczną nazwę. Nazwa `SYS_C004807` została wygenerowana przez bazę danych (automatycznie, więc w innej bazie danych może być inna). To są więzy, których wcześniej nie nazwaliśmy podczas tworzenia.

 **Uwaga** Wysyłając zapytania do perspektywy `all_constraints`, możemy uzyskać informacje o wszystkich więzach, do których mamy dostęp.

Pobieranie informacji o więzach dla kolumny

Wysyłając zapytania do perspektywy `user_cons_columns`, możemy uzyskać informacje o więzach dla kolumn. Tabela 11.5 zawiera opis niektórych kolumn tej perspektywy.

Tabela 11.5. Wybrane kolumny perspektywy `user_cons_columns`

Kolumna	Typ	Opis
owner	VARCHAR2(128)	Właściciel więzów
constraint_name	VARCHAR2(128)	Nazwa więzów
table_name	VARCHAR2(128)	Nazwa tabeli, dla której są zdefiniowane więzy
column_name	VARCHAR2(4000)	Nazwa kolumny, dla której są zdefiniowane więzy

Poniższy przykład pobiera z perspektywy `user_cons_columns` kolumny `constraint_name` i `column_name` dotyczące tabeli `order_status2`:

```
COLUMN column_name FORMAT a15
SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'ORDER_STATUS2'
ORDER BY constraint_name;
```

CONSTRAINT_NAME	COLUMN_NAME
ORDER_STATUS2_ID_CK	ID
ORDER_STATUS2_PK	ID
ORDER_STATUS2_STATUS_CK	STATUS
ORDER_STATUS2_STATUS_NN	STATUS
ORDER_STATUS2_STATUS_UQ	STATUS
SYS_C009710	LAST_MODIFIED

Kolejne zapytanie łączy perspektywy `user_constraints` i `user_cons_columns` w celu pobrania kolumn `column_name`, `constraint_name`, `constraint_type` i `status`:

```
SELECT ucc.column_name, ucc.constraint_name, uc.constraint_type, uc.status
FROM user_constraints uc, user_cons_columns ucc
WHERE uc.table_name = ucc.table_name
AND uc.constraint_name = ucc.constraint_name
AND ucc.table_name = 'ORDER_STATUS2'
ORDER BY ucc.constraint_name;
```

COLUMN_NAME	CONSTRAINT_NAME	C	STATUS
ID	ORDER_STATUS2_ID_CK	C	ENABLED

ID	ORDER_STATUS2_PK	P ENABLED
STATUS	ORDER_STATUS2_STATUS_CK	C ENABLED
STATUS	ORDER_STATUS2_STATUS_NN	C DISABLED
STATUS	ORDER_STATUS2_STATUS_UQ	U ENABLED
LAST_MODIFIED	SYS_CO09710	C ENABLED



Wysyłając zapytania do perspektywy `all_cons_columns`, możemy uzyskać informacje o wszystkich więzach dla kolumn, do których mamy dostęp.

Zmienianie nazwy tabeli

Do zmieniania nazwy tabeli służy instrukcja `RENAME`. Poniższy przykład zmienia nazwę tabeli `order_status2` na `order_state`:

```
RENAME order_status2 TO order_state;
```

Kolejny przykład przywraca pierwotną nazwę tabeli:

```
RENAME order_state TO order_status2;
```

Dodawanie komentarza do tabeli

Komentarz pomaga zapamiętać przeznaczenie tabeli lub kolumny. Do dodawania komentarzy służy instrukcja `COMMENT`. W poniższym przykładzie komentarz jest dodawany do tabeli `order_status2`:

```
COMMENT ON TABLE order_status2 IS
'W order_status2 składowany jest stan zamówienia';
```

Kolejny przykład dodaje komentarz do kolumny `last_modified`:

```
COMMENT ON COLUMN order_status2.last_modified IS
'W last_modified składowana jest data i godzina ostatniej modyfikacji zamówienia';
```

Pobieranie komentarzy do tabeli

Komentarze do tabel można pobrać z perspektywy `user_tab_comments`:

```
SELECT *
FROM user_tab_comments
WHERE table_name = 'ORDER_STATUS2';
```

TABLE_NAME	TABLE_TYPE
-----	-----
COMMENTS	-----
-----	-----
ORDER_STATUS2	TABLE
W order_status2 składowany jest stan zamówienia	

Pobieranie komentarzy do kolumn

Komentarze do kolumn można pobrać z perspektywy `user_col_comments`:

```
SELECT *
FROM user_col_comments
WHERE table_name = 'ORDER_STATUS2';
```

TABLE_NAME	COLUMN_NAME
-----	-----
COMMENTS	-----
-----	-----
ORDER_STATUS2	ID
ORDER_STATUS2	STATUS
ORDER_STATUS2	LAST_MODIFIED
W last_modified składowana jest data i godzina ostatniej modyfikacji zamówienia	

Obcinanie tabeli

Do obcinania tabeli służy instrukcja TRUNCATE. Usuwa ona *wszystkie* wiersze z tabeli, a także resetuje obszar składowania dla tabeli. Poniższy przykład przycina tabelę `order_status2`:

```
TRUNCATE TABLE order_status2;
```



Wskazówka Jeżeli konieczne jest usunięcie wszystkich wierszy z tabeli, należy użyć instrukcji TRUNCATE zamiast DELETE. Pierwsza resetuje bowiem obszar składowania dla nowej tabeli, przygotowując go do przyjęcia nowych wierszy, i nie wymaga żadnej przestrzeni wycofywania w bazie danych. Nie trzeba także uruchamiać instrukcji COMMIT, aby utrzymać usunięcie. Przestrzeń wycofywania jest obszarem wykorzystywanym przez oprogramowanie do zapisywania zmian w bazie danych.

Nowością w Oracle Database 12c jest klauzula CASCADE dla instrukcji TRUNCATE, która obciną wybieraną tabelę, wszystkie tabele od niej zależne, tabele zależne od tabel zależnych itd. Wszystkie tabele, które odwołują się do wskazanej tabeli z więzami ON DELETE CASCADE, zostaną obcięte. Na przykład jeśli miałbyś tabelę o nazwie `tabela_nadrzędna`, to instrukcja `TRUNCATE TABLE tabela_nadrzędna CASCADE` obecnie wszystkie tabele podległe, tabele podległe do tych podległych itd.

Usuwanie tabeli

Do usuwania tabeli służy instrukcja `DROP TABLE`. Poniższy przykład usuwa tabelę `order_status2`:

```
DROP TABLE order_status2;
```

Typy `BINARY_FLOAT` i `BINARY_DOUBLE`

W Oracle Database 10g wprowadzono dwa nowe typy danych: `BINARY_FLOAT` i `BINARY_DOUBLE`. Typ `BINARY_FLOAT` wykorzystuje 32-bitową liczbę zmiennoprzecinkową o pojedynczej precyzyji, a `BINARY_DOUBLE` — 64-bitową liczbę zmiennoprzecinkową o podwójnej precyzyji. Te nowe typy danych opierają się na standardzie IEEE (Institute of Electrical and Electronics Engineers) dla binarnej arytmetyki zmiennoprzecinkowej.

Zalety typów `BINARY_FLOAT` i `BINARY_DOUBLE`

Typy `BINARY_FLOAT` i `BINARY_DOUBLE` są uzupełnieniem typu `NUMBER`, ale mają w stosunku do niego kilka zalet:

- **Posiadają większy zakres reprezentowanych liczb:** obsługują liczby znacznie większe i znacznie mniejsze niż te, które mogą być przechowywane przez typ `NUMBER`.
- **Umożliwiają szybsze wykonywanie obliczeń.** Jest tak, ponieważ operacje na nich są zwykle wykonywane bezpośrednio przez sprzęt, a dane typu `NUMBER` muszą być przekonwertowane przez oprogramowanie przed rozpoczęciem obliczeń.
- **Działania arytmetyczne wykonywane na tych typach danych są zamknięte,** co oznacza, że zwartana jest albo liczba, albo specjalna wartość. Jeżeli na przykład podzielimy `BINARY_FLOAT` przez inną `BINARY_FLOAT`, zostanie zwrócona `BINARY_FLOAT`.
- **Dokładniejsze zaokrąglanie.** Do reprezentacji liczb w tych typach danych jest stosowany system dwójkowy (o podstawie 2), a w typie `NUMBER` — dziesiętny (o podstawie 10). Podstawa wykorzystywana do reprezentowania liczby wpływa na sposób jej zaokrąglania. Na przykład dziesiętna liczba zmiennoprzecinkowa jest zaokrąglana do najbliższego miejsca dziesiętnego, ale dwójkowa liczba zmiennoprzecinkowa jest zaokrąglana do najbliższego miejsca binarnego.
- **Wymagają mniej przestrzeni składowania:** odpowiednio 4 i 8 bajtów przestrzeni, a typ `NUMBER` może zajmować nawet 22 bajty.



Wskazówka Jeżeli opracowujesz system wymagający wielu obliczeń numerycznych, do reprezentacji liczb należy używać typów `BINARY_FLOAT` i `BINARY_DOUBLE`.

Użycie typów BINARY_FLOAT i BINARY_DOUBLE w tabeli

Poniższa instrukcja tworzy tabelę o nazwie `binary_test`, zawierającą kolumnę typu `BINARY_FLOAT` i `BINARY_DOUBLE`:

```
CREATE TABLE binary_test (
    bin_float BINARY_FLOAT,
    bin_double BINARY_DOUBLE
);
```



W katalogu `SQL` znajduje się skrypt o nazwie `oracle_10g_examples.sql`, który tworzy tabelę `binary_test` w schemacie `store`. Ten skrypt wykonuje również instrukcje `INSERT`, prezentowane w tym podrozdziale. Można go uruchomić, jeżeli korzysta się z Oracle Database 10g lub nowszej.

Poniższa instrukcja wstawia wiersz do tabeli `binary_test`:

```
INSERT INTO binary_test (
    bin_float, bin_double
) VALUES (
    39.5f, 15.7d
);
```

`f` wskazuje, że liczba jest typu `BINARY_FLOAT`, a `d` — że typu `BINARY_DOUBLE`¹.

Wartości specjalne

Poniższa tabela pokazuje specjalne wartości, których można użyć z typami `BINARY_FLOAT` i `BINARY_DOUBLE`.

Wartość specjalna	Opis
<code>BINARY_FLOAT_NAN</code>	Nie liczba (ang. <i>not a number</i> , <code>NaN</code>) dla typu <code>BINARY_FLOAT</code>
<code>BINARY_FLOAT_INFINITY</code>	Nieskończoność (ang. <i>infinity</i> , <code>INF</code>) dla typu <code>BINARY_FLOAT</code>
<code>BINARY_DOUBLE_NAN</code>	Nie liczba (ang. <i>not a number</i> , <code>NaN</code>) dla typu <code>DOUBLE_FLOAT</code>
<code>BINARY_DOUBLE_INFINITY</code>	Nieskończoność (ang. <i>infinity</i> , <code>INF</code>) dla typu <code>DOUBLE_FLOAT</code>

W kolejnym przykładzie do tabeli `binary_test` wstawiane są wartości `BINARY_FLOAT_INFINITY` i `BINARY_DOUBLE_INFINITY`:

```
INSERT INTO binary_test (
    bin_float, bin_double
) VALUES (
    BINARY_FLOAT_INFINITY, BINARY_DOUBLE_INFINITY
);
```

Poniższe zapytanie pobiera wiersze z tabeli `binary_test`:

```
SELECT *
FROM binary_test;
```

BIN_FLOAT	BIN_DOUBLE
3,95E+001	1,57E+001
Inf	Inf

Użycie kolumn DEFAULT ON NULL

Nowością w Oracle Database 12c jest klauzula `DEFAULT ON NULL`. Ta klauzula przypisuje domyślną wartość w kolumnie, gdy instrukcja `INSERT` wstawia do niej wartość `NULL`.

¹ W tym przypadku jako separator dziesiętny jest stosowana kropka. Przecinek służy do oddzielania kolejnych wartości — przyp. tłum.

Poniższy przykład tworzy tabelę o nazwie `purchases_default_null`. Kolumna `quantity` ma wstawianą wartość 1, gdy w instrukcji `INSERT` wstawiana jest wartość `NULL`.

```
CREATE TABLE purchases_default_null (
    product_id INTEGER
        CONSTRAINT purch_default_fk_products
        REFERENCES products(product_id),
    customer_id INTEGER
        CONSTRAINT purch_default_fk_customers
        REFERENCES customers(customer_id),
    quantity INTEGER DEFAULT ON NULL 1 NOT NULL,
    CONSTRAINT purch_default_pk PRIMARY KEY (product_id, customer_id)
);
```

Poniższa instrukcja `INSERT` dodaje do tabeli wiersz, pomijając wartość dla kolumny `quantity`:

```
INSERT INTO purchases_default_null (
    product_id, customer_id
) VALUES (
    5, 4
);
```

Poniższa instrukcja `INSERT` dodaje do tabeli wiersz, wstawiając do kolumny `quantity` wartość `NULL`:

```
INSERT INTO purchases_default_null (
    product_id, customer_id, quantity
) VALUES (
    6, 5, NULL
);
```

Poniższa instrukcja `INSERT` dodaje wiersz do tabeli, wstawiając do kolumny `quantity` wartość 3:

```
INSERT INTO purchases_default_null (
    product_id, customer_id, quantity
) VALUES (
    7, 2, 3
);
```

Poniższa instrukcja pobiera wiersze z tabeli:

```
SELECT *
FROM purchases_default_null
ORDER BY product_id;
```

PRODUCT_ID	CUSTOMER_ID	QUANTITY
5	4	1
6	5	1
7	2	3

Dwa pierwsze wiersze w kolumnie `quantity` mają wartość 1, która jest wartością domyślną ustawioną przez klauzulę `DEFAULT ON NULL` w definicji tabeli. Trzeci wiersz ma w kolumnie `quantity` wartość 3, która była jawnie wpisana w instrukcji `INSERT`.

Kolumny niewidoczne

Nowością w Oracle Database 12c jest możliwość określania widoczności kolumn w tabeli. Aby oznaczyć kolumnę jako widoczną, używamy klauzuli `VISIBL`, a w przypadku kolumny niewidocznej używamy klauzuli `INVISIBLE`. Jeżeli nie użyjemy żadnej z powyższych klauzul, kolumna zostanie domyślnie uznana za widoczną.

Poniższy przykład tworzy tabelę `employees_hidden_example`. Kolumna `salary` jest oznaczona jako `INVISIBLE`. Kolumna `title` jest jawnie oznaczona jako `VISIBL`. Inne kolumny są widoczne domyślnie.

```
CREATE TABLE employees_hidden_example (
    employee_id INTEGER CONSTRAINT employees_hidden_example PRIMARY KEY,
    manager_id INTEGER,
```

```

first_name VARCHAR2(10) NOT NULL,
last_name VARCHAR2(10) NOT NULL,
title VARCHAR2(20) VISIBLE,
salary NUMBER(6, 0) INVISIBLE
);

```

Domyślnie kolumny niewidoczne nie są pokazywane przez instrukcję DESCRIBE. W poniższym przykładzie kolumna salary nie jest wyświetlona:

```
DESCRIBE employees_hidden_example;
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(38)
MANAGER_ID		NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
TITLE		VARCHAR2(20)

Aby instrukcja DESCRIBE pokazała niewidoczne kolumny, należy użyć instrukcji SET COLINVISIBLE ON. Aby ukryć te kolumny, należy użyć instrukcji SET COLINVISIBLE OFF. Domyślnie kolumny takie nie są pokazywane. Poniższy przykład wykorzystuje instrukcję SET COLINVISIBLE ON i wyświetla opis tabeli employees_hidden_example:

```
SET COLINVISIBLE ON
DESCRIBE employees_hidden_example;
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(38)
MANAGER_ID		NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
TITLE		VARCHAR2(20)
SALARY (INVISIBLE)		NUMBER(6)

W wynikach widzimy kolumnę salary.

Poniższa instrukcja INSERT zwraca błąd, ponieważ niewidoczna kolumna salary nie może być w ten sposób wypełniona:

```
INSERT INTO employees_hidden_example VALUES (
  1, 1, 'Jan', 'Szmał', 'CEO', 250000
);
```

```
INSERT INTO employees_hidden_example VALUES (
  *
```

ERROR at line 1:

ORA-00913: za duża liczba wartości

Kolumna niewidoczna musi być jawnie wymieniona na liście kolumn do wypełnienia, jak pokazane jest w poniższej poprawnej instrukcji INSERT:

```
INSERT INTO employees_hidden_example (
  employee_id, manager_id, first_name, last_name, title, salary
) VALUES (
  1, 1, 'Jan', 'Szmał', 'CEO', 250000
);
```

Poniższe zapytanie pobiera wiersz z tabeli. Można zauważyć, że kolumna salary nie jest wyświetlona wśród wyników zapytania:

```
SELECT *
FROM employees_hidden_example;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE
1	1	Jan	Szmał	CEO

Aby wyświetlić zawartość niewidocznej kolumny, należy jawnie tego zażądać. W poniższym przykładzie kolumna `salary` jest jawnie wymieniona na ostatnim miejscu listy kolumn:

```
SELECT employee_id, manager_id, first_name, last_name, title, salary
FROM employees_hidden_example;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1	1	Jan	Szmal	CEO	250000

Można zmodyfikować tabelę, oznaczając widoczną kolumnę jako niewidoczną lub niewidoczną jako widoczną. W poniższym przykładzie kolumna `salary` jest oznaczana jako widoczna, a kolumna `title` jako niewidoczna:

```
ALTER TABLE employees_hidden_example MODIFY (
    title INVISIBLE,
    salary VISIBLE
);
```

Poniższe zapytanie pobiera wiersz z tabeli. Można zauważyć, że kolumna `title` nie jest wyświetlana:

```
SELECT *
FROM employees_hidden_example;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	SALARY
1	1	Jan	Szmal	250000

W dalszej części książki będzie pokazane, że można używać widocznych i niewidocznych kolumn również w widokach.

Na tym zakończymy omówienie tabel. W kolejnym podrozdziale omówimy sekwencje.

Sekwencje

Sekwencja jest elementem bazy danych, generującym sekwencje liczb całkowitych. Liczby całkowite generowane przez sekwencje są zwykle używane do wprowadzania danych do liczbowej kolumny klucza głównego. Z tego podrozdziału dowiesz się, jak:

- tworzyć sekwencje,
- pobierać informacje o nich ze słownika danych,
- używać sekwencji na wiele różnych sposobów,
- wypełniać klucz główny za pomocą sekwencji,
- określać domyślną wartość kolumny za pomocą sekwencji,
- wykorzystywać kolumny typu `IDENTITY`,
- modyfikować sekwencje,
- usuwać sekwencje.

Tworzenie sekwencji

Do tworzenia sekwencji służy instrukcja `CREATE SEQUENCE`, która ma następującą składnię:

```
CREATE SEQUENCE nazwa_sekwencji
[START WITH liczba_pocz]
[INCREMENT BY liczba_inkr]
[ { MAXVALUE maks_wart | NOMAXVALUE } ]
[ { MINVALUE min_wart | NOMINVALUE } ]
[ { CYCLE | NOCYCLE } ]
[ { CACHE liczba_w_buf | NOCACHE } ]
[ { ORDER | NOORDER } ];
```

gdzie:

- *nazwa_sekwencji* jest nazwą sekwencji.
- *liczba_pocz* jest liczbą całkowitą rozpoczynającą sekwencję (domyślnie 1).
- *liczba_inkr* jest liczbą całkowitą zwiększającą sekwencję (domyślnie 1). Wartość absolutna *liczba_inkr* musi być mniejsza niż różnica między *maks_wart* i *min_wart*.
- *maks_wart* jest maksymalną liczbą całkowitą w sekwencji; *maks_wart* musi być równa wartości *liczba_pocz* lub większa od niej, a także większa od *min_wart*.
- **NOMAXVALUE** określa, że maksymalną wartością sekwencji rosnącej będzie 10^{27} , a w sekwencji malejącej będzie to -1 . Jest to ustawienie domyślne.
- *min_wart* jest minimalną liczbą całkowitą w sekwencji; musi być równa wartości *liczba_pocz* lub mniejsza od niej oraz mniejsza od *maks_wart*.
- **NOMINVALUE** określa, że w sekwencji rosnącej minimalną liczbą będzie 1 , a w sekwencji malejącej będzie to -10^{26} . Jest to ustawienie domyślne.
- **CYCLE** oznacza, że sekwencja będzie generowała liczby całkowite nawet po osiągnięciu wartości minimalnej lub maksymalnej. Jeżeli sekwencja rosnąca osiągnie wartość maksymalną, kolejną wygenerowaną liczbą będzie wartość minimalna. Jeżeli sekwencja malejąca osiągnie wartość minimalną, kolejną wygenerowaną liczbą będzie wartość maksymalna.
- **NOCYCLE** oznacza, że po osiągnięciu wartości minimalnej lub maksymalnej sekwencja nie będzie generowała dalszych liczb całkowitych. Jest to ustawienie domyślne.
- **CACHE** oznacza przechowywanie liczb w pamięci podręcznej. Powoduje to alokowanie z wyprzedzeniem liczb dla sekwencji.
- *liczba_buf* jest liczbą liczb całkowitych przechowywanych w pamięci. Domyślnie buforowanych jest 20 liczb całkowitych. Muszą być buforowane przynajmniej dwie. Maksymalna liczba liczb całkowitych, która może być przechowana w pamięci podręcznej, jest określona wzorem $\text{CEIL}(\text{maks_wart} - \text{min_wart})/\text{ABS}(\text{liczba_inkr})$.
- **NOCACHE** oznacza brak przechowywania w pamięci podręcznej. Wówczas oprogramowanie danych nie będzie alokowało wartości sekwencji z wyprzedzeniem, co zapobiega powstawaniu luk liczbowych w sekwencji, ale zmniejsza wydajność. Luki powstają, ponieważ wartości przechowywane w pamięci podręcznej nie są zapisywane przy zamknięciu bazy danych. Jeżeli zostaną pominione opcje **CACHE** i **NOCACHE**, w pamięci podręcznej będzie składowanych 20 liczb całkowitych.
- **ORDER** gwarantuje, że liczby całkowite będą generowane w kolejności żądań. Opcja ta jest zwykle stosowana, jeżeli używana jest technologia Real Application Cluster, konfigurowana i zarządzana przez administratorów baz danych. W tej technologii jest wykorzystywanych wiele serwerów baz danych współdzielących pamięć. Może zwiększyć wydajność.
- **NOORDER** nie gwarantuje, że liczby całkowite będą generowane zgodnie z kolejnością żądań. Jest to ustawienie domyślne.

W poniższym przykładzie łączymy się jako użytkownik **store** i tworzymy sekwencję o nazwie **s_test**:

```
CONNECT store/store_password
CREATE SEQUENCE s_test;
```

Ponieważ w instrukcji **CREATE SEQUENCE** pominięto parametry opcjonalne, zostały użyte domyślne ustawienia. To oznacza, że *liczba_pocz* oraz *liczba_inkr* mają wartości 1.

W kolejnym przykładzie jest tworzona sekwencja **s_test2** przy określonych parametrach opcjonalnych:

```
CREATE SEQUENCE s_test2
START WITH 10 INCREMENT BY 5
MINVALUE 10 MAXVALUE 20
CYCLE CACHE 2 ORDER;
```

W ostatnim przykładzie jest tworzona sekwencja **s_test3**, rozpoczynająca się od 10 i licząca w dół:

```
CREATE SEQUENCE s_test3
START WITH 10 INCREMENT BY -1
MINVALUE 1 MAXVALUE 10
CYCLE CACHE 5;
```

Pobieranie informacji o sekwencjach

Informacje o sekwencjach można pobrać z perspektywy `user_sequences`. W tabeli 11.6 opisano jej kolumny.

Tabela 11.6. Wybrane kolumny perspektywy `user_sequences`

Kolumna	Typ	Opis
<code>sequence_name</code>	<code>VARCHAR2(128)</code>	Nazwa sekwencji
<code>min_value</code>	<code>NUMBER</code>	Wartość minimalna
<code>max_value</code>	<code>NUMBER</code>	Wartość maksymalna
<code>increment_by</code>	<code>NUMBER</code>	Wartość, o jaką sekwencja będzie zwiększana lub zmniejszana
<code>cycle_flag</code>	<code>VARCHAR2(1)</code>	Określa, czy sekwencja jest cykliczna (Y lub N)
<code>order_flag</code>	<code>VARCHAR2(1)</code>	Określa, czy sekwencja jest uporządkowana (Y lub N)
<code>cache_size</code>	<code>NUMBER</code>	Liczba wartości sekwencji przechowywanych w pamięci
<code>last_number</code>	<code>NUMBER</code>	Ostatnia liczba wygenerowana lub zapisana w pamięci przez sekwencję

Poniższy przykład pobiera z perspektywy `user_sequences` szczegółowe informacje o sekwencjach:

```
COLUMN sequence_name FORMAT a13
SELECT
    sequence_name, min_value, max_value, increment_by,
    cycle_flag, order_flag, cache_size, last_number
FROM user_sequences
ORDER BY sequence_name;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	C	O	CACHE_SIZE	LAST_NUMBER
S_TEST	1	1,0000E+28	1	N	N	20	1
S_TEST2	10	20	5	Y	Y	2	10
S_TEST3	1	10	-1	Y	N	5	10



Wysyłając zapytania do perspektywy `all_sequences`, możemy uzyskać informacje o wszystkich sekwencjach, do których mamy dostęp.

Używanie sekwencji

Sekwencja generuje ciąg liczb i zawiera dwie pseudokolumny o nazwach `currval` i `nextval` używanych do pobierania bieżącej i kolejnej wartości z sekwencji.

Przed pobraniem bieżącej wartości sekwencja musi zostać zainicjalizowana przez pobranie następnej wartości. Gdy pobierzemy `s_test.nextval`, sekwencja zostanie zainicjalizowana z liczbą 1. Poniżejsze zapytanie pobiera `s_test.nextval`. Należy zauważyć, że w klauzuli `FROM` została użyta tabela `dual`:

```
SELECT s_test.nextval
FROM dual;
```

```
NEXTVAL
-----
1
```

Pierwszą wartością w sekwencji `s_test` jest 1. Po zainicjalizowaniu sekwencji można z niej pobrać bieżącą wartość przez pobranie `currval`. Na przykład:

```
SELECT s_test.currval
FROM dual;
```

```
CURRVAL
```

```
-----
```

```
1
```

Po pobraniu `currval`, `nextval` pozostaje niezmieniona. Ta wartość zmienia się jedynie wtedy, gdy po bierzemy `nextval` w celu uzyskania kolejnej wartości. Poniższy przykład pobiera `s_test.nextval` i `s_test.currval`, obie wartości wynoszą 2:

```
SELECT s_test.nextval, s_test.currval
FROM dual;
```

```
NEXTVAL    CURRVAL
```

```
-----
```

```
2          2
```

Pobranie `s_test.nextval` zwraca kolejną wartość z sekwencji, czyli 2; `s_test.currval` również wynosi 2.

Kolejny przykład inicjalizuje sekwencję `s_test2` przez pobranie `s_test2.nextval`; pierwszą liczbą w sekwencji jest 10:

```
SELECT s_test2.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
10
```

Maksymalną wartością w `s_test2` jest 20, a sekwencja ta została utworzona z opcją CYCLE, co oznacza, że po osiągnięciu maksimum (20) sekwencja ponownie rozpocznie cykl od 10:

```
SELECT s_test2.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
15
```

```
SELECT s_test2.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
20
```

```
SELECT s_test2.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
10
```

Sekwencja `s_test3` rozpoczyna się od 10 i generuje coraz mniejsze liczby aż do 1:

```
SELECT s_test3.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
10
```

```
SELECT s_test3.nextval
FROM dual;
```

```
NEXTVAL
```

```
-----
```

```
9
```

```
SELECT s_test3.nextval
```

```
FROM dual;
NEXTVAL
-----
8
```

Wypełnianie klucza głównego z użyciem sekwencji

Sekwencje są przydatne do wypełniania kolumn klucza głównego liczbami całkowitymi. Poniższa instrukcja ponownie tworzy tabelę `order_status2`:

```
CREATE TABLE order_status2 (
    id INTEGER CONSTRAINT order_status2_pk PRIMARY KEY,
    status VARCHAR2(10),
    last_modified DATE DEFAULT SYSDATE
);
```

Ta instrukcja tworzy sekwencję `s_order_status2` (zostanie ona użyta do wypełnienia kolumny `order_status2.id`):

```
CREATE SEQUENCE s_order_status2 NOCACHE;
```

 **Wskazówka** Jeżeli sekwencja jest używana do wypełniania klucza głównego, zwykle powinno się stosować opcję `NOCACHE` w celu uniknięcia luk w ciągu liczb (luki pojawiają się, ponieważ wartości przechowywane w pamięci podręcznej nie są zapisywane przy zamknięciu bazy danych). Użycie tej opcji zmniejsza jednak wydajność. Jeżeli jesteśmy pewni, że w wartościach klucza głównego mogą występować luki, możemy użyć opcji `CACHE`.

Poniższa instrukcja `INSERT` wstawia wiersze do tabeli `order_status2`. Wartość dla kolumny `id` jest ustawiana z użyciem `s_order_status2.nextval` (co zwraca 1 w przypadku pierwszej instrukcji `INSERT` i 2 w przypadku drugiej):

```
INSERT INTO order_status2 (
    id, status, last_modified
) VALUES (
    s_order_status2.nextval, 'ZŁOŻONE', '06/01/01'
);

INSERT INTO order_status2 (
    id, status, last_modified
) VALUES (
    s_order_status2.nextval, 'OCZEKUJĄCE', '06/02/01'
);
```

Poniższe zapytanie pobiera wiersze z tabeli `order_status2`. W kolumnie `id` znajdują się pierwsze dwie wartości (1 i 2) z sekwencji `s_order_status2`:

```
SELECT *
FROM order_status2;
```

ID	STATUS	LAST_MOD
1	ZŁOŻONE	06/01/01
2	OCZEKUJĄCE	06/02/01

Określanie domyślnej wartości kolumny za pomocą sekwencji

Nowością w Oracle Database 12c jest możliwość określania domyślnej wartości kolumny za pomocą sekwencji.

Poniższy przykład tworzy sekwencję o nazwie `s_default_value_for_column`:

```
CREATE SEQUENCE s_default_value_for_column;
```

Poniższy przykład tworzy tabelę o nazwie `test_with_sequence` i określa, że domyślną wartością kolumny `sequence_value` będzie kolejna wartość z sekwencji:

```
CREATE TABLE test_with_sequence (
    id INTEGER CONSTRAINT test_with_sequence_pk PRIMARY KEY,
    sequence_value INTEGER DEFAULT s_default_value_for_column.NEXTVAL
);
```

Poniższe instrukcje INSERT dodają wiersze do tabeli:

```
INSERT INTO test_with_sequence (id) VALUES (1);
INSERT INTO test_with_sequence (id) VALUES (2);
INSERT INTO test_with_sequence (id) VALUES (4);
```

Kolejne zapytanie pobiera wiersze z tabeli. Można zauważyć, że kolumna `sequence_value` wypełniona jest pierwszymi trzema wartościami wygenerowanymi przez sekwencję, czyli 1, 2 i 3.

```
SELECT *
FROM test_with_sequence;
```

ID	SEQUENCE_VALUE
1	1
2	2
4	3

Przy ustawianiu domyślnej wartości kolumny za pomocą sekwencji należy pamiętać o dwóch sprawach:

- Użytkownicy dodający wiersze do tabeli muszą mieć uprawnienia do używania instrukcji `INSERT` na tabeli oraz do używania instrukcji `SELECT` na sekwencji.
- Jeżeli sekwencja zostanie usunięta, kolejne instrukcje `INSERT` zwrócią komunikat o błędzie.

Kolumny typu IDENTITY

Nowością w Oracle Database 12c są kolumny typu `IDENTITY`. Wartość takiej kolumny jest określana za pomocą wyrażenia generującego sekwencję.

W poniższym przykładzie utworzona jest tabela `test_with_identity` z tabelą `identity_value` mającą określoną jako wartość domyślną kolejną wartość z sekwencji:

```
CREATE TABLE test_with_identity (
    id INTEGER CONSTRAINT test_with_identity_pk PRIMARY KEY,
    identity_value INTEGER GENERATED BY DEFAULT AS IDENTITY (
        START WITH 5 INCREMENT BY 2
    )
);
```

Poniższa instrukcja `INSERT` dodaje wiersze do tabeli:

```
INSERT INTO test_with_identity (id) VALUES (1);
INSERT INTO test_with_identity (id) VALUES (2);
INSERT INTO test_with_identity (id) VALUES (4);
```

Kolejne zapytanie pobiera wiersze z tabeli. Można zauważyć, że w kolumnie `identity_value` znajdują się trzy pierwsze wartości wygenerowane przez sekwencję, czyli 5, 7 i 9.

```
SELECT *
FROM test_with_identity;
```

ID	IDENTITY_VALUE
1	5
2	7
4	9

Modyfikowanie sekwencji

Do modyfikowania sekwencji służy instrukcja `ALTER SEQUENCE`. Oto ograniczenia dotyczące tych operacji:

- nie można zmienić wartości początkowej sekwencji,
- minimalna wartość nie może być większa niż bieżąca wartość sekwencji,

- maksymalna wartość nie może być mniejsza od bieżącej wartości sekwencji.

Poniższy przykład zmienia sekwencję s_test tak, aby jej wartości były zwiększane o 2:

```
ALTER SEQUENCE s_test
INCREMENT BY 2;
```

Po wykonaniu tej instrukcji nowe wartości generowane przez s_test będą zwiększane o 2. Jeżeli s_test.currval wynosi 2, to s_test.nextval będzie wynosiło 4:

```
SELECT s_test.currval
FROM dual;
```

```
CURRVAL
-----
2
```

```
SELECT s_test.nextval
FROM dual;
```

```
NEXTVAL
-----
4
```

Usuwanie sekwencji

Do usuwania sekwencji służy instrukcja `DROP SEQUENCE`. Poniższy przykład usuwa sekwencję s_test3:

```
DROP SEQUENCE s_test3;
```

Na tym zakończymy omawianie sekwencji. W kolejnym podrozdziale zajmiemy się indeksami.

Indeksy

Jeżeli szukamy w książce informacji, możemy ją całą przeczytać albo użyć indeksu, żeby odnaleźć odpowiedni fragment. W założeniach indeks tabeli bazy danych przypomina indeks książki, z tą różnicą, że indeksy bazodanowe są używane do wyszukiwania konkretnych wierszy w tabeli. Wada indeksów polega na tym, że gdy do tabeli jest wstawiany wiersz, jest wymagany dodatkowy czas konieczny do aktualizacji indeksu o nowy wiersz.

Indeks powinien być tworzony na kolumnie, jeżeli pobieramy niewielką liczbę wierszy z tabeli zawierającej ich wiele. Należy się kierować ogólną zasadą:

Indeks powinien być tworzony, jeżeli zapytanie pobiera <=10 procent wszystkich wierszy tabeli.

To oznacza, że kolumna wyznaczona na indeks powinna zawierać duży zakres wartości. Tego typu indeksy nazywamy indeksami **B-drzewo** — nazwa pochodzi od struktury drzewa danych, stosowanej w informatyce. Dobrym kandydatem na indeks B-drzewo jest kolumna zawierająca unikatowe wartości w każdym wierszu (na przykład numer PESEL). Złyim pomysłem na tego typu indeks jest kolumna zawierająca tylko niewielki zakres wartości (na przykład N, S, E i W lub 1, 2, 3, 4, 5, i 6). Oprogramowanie bazy danych Oracle automatycznie tworzy indeks B-drzewo dla kluczów głównych tabeli oraz kolumn zawartych w więzach unikatowości. W przypadku kolumn zawierających niewielki zakres wartości można zastosować indeks bitmapowy.

Z tego podrozdziału dowiesz się, jak:

- tworzyć indeksy typu B-drzewo,
- tworzyć indeksy oparte na funkcjach,
- pobierać informacje o indeksie ze słownika danych,
- modyfikować indeks,
- usuwać indeks,
- tworzyć indeksy bitmapowe.

Tworzenie indeksu typu B-drzewo

Do tworzenia indeksu typu B-drzewo służy instrukcja `CREATE INDEX`, której uproszczona składnia jest następująca:

```
CREATE [UNIQUE] INDEX nazwa_indeksu ON
nazwa_tabeli(nazwa_kolumny[, nazwa_kolumny ...])
TABLESPACE przestrzen_tab;
```

gdzie:

- `UNIQUE` oznacza, że wartości w indeksowanej kolumnie muszą być unikatowe.
- `nazwa_indeksu` jest nazwą indeksu.
- `nazwa_tabeli` jest nazwą tabeli w bazie danych.
- `nazwa_kolumny` jest indeksowaną kolumną. Indeks można utworzyć na wielu kolumnach (taki indeks nazywamy **indeksem złożonym**).
- `przestrzen_tab` jest przestrzenią tabel dla indeksu. Jeżeli ten parametr nie zostanie określony, indeks będzie składowany w domyślnej przestrzeni tabel użytkownika.

 **Wskazówka** Ze względów wydajnościowych indeksy powinny być składowane w innej przestrzeni tabel niż tabele. Dla uproszczenia w przykładach prezentowanych w tym rozdziale indeksy wykorzystują domyślną przestrzeń tabel. W produkcyjnej bazie danych administrator powinien utworzyć oddzielne przestrzenie tabel dla indeksów i tabel.

Omówię teraz procedurę tworzenia indeksu typu B-drzewo dla kolumny `customers.last_name`. Zasadzamy, że tabela `customers` zawiera bardzo dużo wierszy, które często pobieramy za pomocą zapytania podobnego do tego:

```
SELECT customer_id, first_name, last_name
FROM customers
WHERE last_name = 'Brąz';
```

Zakładamy również, że kolumna `last_name` zawiera na tyle unikatowe wartości, że zapytanie wykorzystujące ją w klauzuli `WHERE` będzie zwracało mniej niż 10 procent wierszy z całej tabeli. To oznacza, że kolumna `last_name` może być indeksowana.

Poniższa instrukcja `CREATE INDEX` tworzy indeks `i_customers_last_name` na kolumnie `last_name` tabeli `customers`:

```
CREATE INDEX i_customers_last_name ON customers(last_name);
```

Po utworzeniu indeksu wcześniej prezentowane zapytanie powinno zostać ukończone w krótszym czasie.

Używając indeksu unikatowego, możemy wymusić unikatowość wartości w kolumnie. Na przykład poniższa instrukcja tworzy unikatowy indeks `i_customers_phone` na kolumnie `customers.phone`:

```
CREATE UNIQUE INDEX i_customers_phone ON customers(phone);
```

Możemy również utworzyć indeks złożony oparty na wielu kolumnach. Na przykład poniższa instrukcja tworzy indeks złożony `i_employees_first_last_name`, oparty na kolumnach `first_name` i `last_name` tabeli `employees`:

```
CREATE INDEX i_employees_first_last_name ON
employees(first_name, last_name);
```

Tworzenie indeksów opartych na funkcjach

W poprzednim podrozdziale utworzyliśmy indeks `i_customers_last_name`. Założmy, że uruchamiamy następujące zapytanie:

```
SELECT first_name, last_name
FROM customers
WHERE last_name = UPPER('BRĄZ');
```

Ponieważ w zapytaniu używana jest funkcja — `UPPER()` — indeks `i_customers_last_name` nie jest stosowany. Jeżeli chcemy, aby indeks był oparty na wynikach funkcji, musimy utworzyć indeks oparty na funkcji:

```
CREATE INDEX i_func_customers_last_name
ON customers(UPPER(last_name));
```

Aby indeksy oparte na funkcjach były wykorzystywane, administrator bazy danych musi ustawić parametr inicjalizujący `QUERY_REWRITE_ENABLED` na wartość `true` (domyślnie ustawiona jest wartość `false`). Poniższy przykład ustawia wartość parametru `QUERY_REWRITE_ENABLED` na wartość `true`:

```
CONNECT system/manager
ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;
```

Pobieranie informacji o indeksach

Informacje o indeksach możemy pobrać z perspektywy `user_indexes`. Tabela 11.7 zawiera opis jej niektórych kolumn.

Tabela 11.7. Wybrane kolumny perspektywy `user_indexes`

Kolumna	Typ	Opis
<code>index_name</code>	<code>VARCHAR2(128)</code>	Nazwa indeksu
<code>table_owner</code>	<code>VARCHAR2(128)</code>	Nazwa użytkownika będącego właścicielem tabeli
<code>table_name</code>	<code>VARCHAR2(128)</code>	Nazwa tabeli, dla której został utworzony indeks
<code>uniqueness</code>	<code>VARCHAR2(9)</code>	Okręsła, czy indeks jest unikatowy (UNIQUE lub NONUNIQUE)
<code>status</code>	<code>VARCHAR2(8)</code>	Okręsła, czy indeks jest prawidłowy (VALID lub INVALID)

W poniższym przykładzie łączymy się jako użytkownik `store` i pobieramy z perspektywy `user_indexes` niektóre kolumny dotyczące tabel `customers` i `employees`. Należy zauważyć, że na liście indeksów znajduje się `customers_pk`, czyli unikatowy indeks automatycznie utworzony przez bazę danych dla kolumny `customer_id`, będącej kluczem głównym tabeli `customers`:

```
CONNECT store/store_password
SELECT index_name, table_name, uniqueness, status
FROM user_indexes
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

INDEX_NAME	TABLE_NAME	UNIQUENESS	STATUS
CUSTOMERS_PK	CUSTOMERS	UNIQUE	VALID
EMPLOYEES_PK	EMPLOYEES	UNIQUE	VALID
I_CUSTOMERS_LAST_NAME	CUSTOMERS	NONUNIQUE	VALID
I_CUSTOMERS_PHONE	CUSTOMERS	UNIQUE	VALID
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	NONUNIQUE	VALID
I_FUNC_CUSTOMERS_LAST_NAME	CUSTOMERS	NONUNIQUE	VALID

Informacje o wszystkich indeksach, do których mamy dostęp, można uzyskać, wysyłając zapytania do perspektywy `all_indexes`.

Pobieranie informacji o indeksach kolumny

Wysyłając zapytania do perspektywy `user_ind_columns`, możemy pobrać informacje o indeksach na kolumnie. W tabeli 11.8 opisano niektóre kolumny tej perspektywy.

Poniższe zapytanie pobiera z perspektywy `user_ind_columns` niektóre kolumny tabel `customers` i `employees`:

```
COLUMN table_name FORMAT a15
COLUMN column_name FORMAT a15
```



Tabela 11.8. Wybrane kolumny perspektywy user_ind_columns

Kolumna	Typ	Opis
index_name	VARCHAR2(128)	Nazwa indeksu
table_name	VARCHAR2(128)	Nazwa tabeli, dla której został utworzony indeks
column_name	VARCHAR2(4000)	Nazwa indeksowanej kolumny

```
SELECT index_name, table_name, column_name
FROM user_ind_columns
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

INDEX_NAME	TABLE_NAME	COLUMN_NAME
CUSTOMERS_PK	CUSTOMERS	CUSTOMER_ID
EMPLOYEES_PK	EMPLOYEES	EMPLOYEE_ID
I_CUSTOMERS_LAST_NAME	CUSTOMERS	LAST_NAME
I_CUSTOMERS_PHONE	CUSTOMERS	PHONE
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	FIRST_NAME
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	LAST_NAME
I_FUNC_CUSTOMERS_LAST_NAME	CUSTOMERS	SYS_NC00006\$

 **Uwaga** Wysyłając zapytania do perspektywy all_ind_columns, można pobrać informacje o wszystkich indeksach, do których mamy dostęp.

Modyfikowanie indeksu

Indeks można zmodyfikować za pomocą instrukcji ALTER INDEX. W poniższym przykładzie jest zmieniana nazwa indeksu i_customers_phone na i_customers_phone_number:

```
ALTER INDEX i_customers_phone RENAME TO i_customers_phone_number;
```

Usuwanie indeksu

Do usuwania indeksów służy instrukcja DROP INDEX. W poniższym przykładzie usuwany jest indeks i_customers_phone_number:

```
DROP INDEX i_customers_phone_number;
```

Tworzenie indeksu bitmapowego

Indeksy bitmapowe są zwykle używane w **hurtowniach danych**, czyli bazach danych zawierających wielkie ilości danych. Organizacje wykorzystują zazwyczaj hurtownie danych do analiz biznesowych takich jak monitorowanie trendów sprzedaży czy badanie zachowań klientów. Dane składowane w hurtowni danych są zwykle odczytywane przez złożone zapytania, ale nie są często modyfikowane. Dane mogą być aktualizowane tylko na koniec dnia, tygodnia lub w innych określonych odstępach czasu.

Kandydatem na indeks bitmapowy jest kolumna, która:

- pojawia się w wielu zapytaniach,
- zawiera niewielki zakres wartości.

Przykładem niewielkich zakresów wartości są:

- N, S, E, W,
- 1, 2, 3, 4, 5, 6,
- „Zamówienie złożone”, „Zamówienie realizowane”, „Zamówienie wysłane”, „Zamówienie anulowane”.

Indeks zawiera wskaźnik do wiersza w tabeli, który zawiera daną wartość klucza indeksu. Jest ona używana do pobrania identyfikatora (ROWID) wiersza tabeli. (Jak zostało to opisane w rozdziale 2., ROWID

jest używany wewnętrznie przez bazę danych do składowania fizycznego położenia wiersza). W indeksie typu B-drzewo lista identyfikatorów wierszy jest składowana dla każdego klucza odpowiadającego wierszom z tą wartością klucza. W tego typu indeksie baza danych składa się z listy wartości klucza z każdym identyfikatorem wiersza, co umożliwia zlokalizowanie wiersza w tabeli.

W indeksie bitmapowym dla każdej wartości klucza jest używana mapa bitowa. Umożliwia ona zlokalizowanie wiersza. Każdy bit odpowiada możliwemu identyfikatorowi wiersza. Jeżeli jest ustawiony, wiersz z danym identyfikatorem zawiera wartość klucza. Funkcja mapująca przekształca pozycję bitu na ROWID.

Poniższe punkty można traktować jako wskazówki przy podejmowaniu decyzji o tworzeniu indeksu bitmapowego:

- Indeksy bitmapowe są zwykle stosowane w tabelach zawierających dużą ilość danych, rzadko modyfikowanych.
- Indeks bitmapowy powinien być tworzony na kolumnach zawierających niewielką liczbę różnych wartości. Jeżeli liczba różnych wartości w kolumnie jest mniejsza niż 1 procent liczby wierszy w tabeli lub jeżeli wartości w kolumnie powtarzają się więcej niż 100 razy, kolumna nadaje się do utworzenia indeksu bitmapowego. Na przykład gdybyśmy mieli tabelę zawierającą milion wierszy, kolumna z 10 000 lub mniejszą liczbą różnych wartości stanowiłaby dobrą podstawę dla indeksu bitmapowego.
- Kolumna z indeksem bitmapowym powinna być często używana w klauzuli WHERE zapytań.

Poniższa instrukcja tworzy indeks bitmapowy na kolumnie `status` tabeli `order_status`:

```
CREATE BITMAP INDEX i_order_status ON order_status(status);
```

 **Uwaga** Ten przykład nie jest najlepszy, ponieważ tabela `order_status` nie zawiera wystarczającej liczby wierszy.

Więcej informacji na temat indeksów bitmapowych można znaleźć w *Oracle Database Performance Tuning Guide* i *Oracle Database Concepts* opublikowanych przez Oracle Corporation. Te książki zawierają również informacje o innych rzadko spotykanych typach indeksów.

Na tym zakończymy omawianie indeksów. W kolejnym podrozdziale omówimy perspektywy.

Perspektywy

Perspektywa jest predefiniowanym zapytaniem jednej lub wielu tabel (zwanych **tabelami bazowymi**). Pobieranie informacji z perspektywy odbywa się w taki sposób jak pobieranie informacji z tabeli. Wystarczy jedynie umieścić nazwę perspektywy w klauzuli `FROM`. W przypadku niektórych perspektyw można wykonywać operacje DML na tabelach bazowych.

 **Uwaga**

W perspektywach nie są składowane wiersze. Są one zawsze składowane w tabelach.

Widzieliśmy już kilka przykładów pobierania informacji z perspektyw, na przykład gdy wybieraliśmy wiersze ze słownika danych, do którego dostęp uzyskuje się za ich pośrednictwem (`user_tables`, `user_sequences` i `user_indexes` są widokami).

Perspektywy mają kilka zalet:

- Umożliwiają umieszczenie złożonego zapytania w perspektywie i przyznanie do niej dostępu użytkownikom. To pozwala ukryć złożoność przed użytkownikami.
- Pozwalają na uniemożliwienie użytkownikom bezpośredniego wysyłania zapytań do tabel bazy danych, przyznając im dostęp jedynie do widoków.
- Umożliwiają przyznanie perspektywie dostępu jedynie do określonych wierszy tabel bazowych, co pozwala na ukrywanie wierszy przed użytkownikami.

Z tego podrozdziału dowiesz się, jak:

- tworzyć perspektywy i używać ich,
- modyfikować perspektywy,
- usuwać je,
- wykorzystywać niewidoczne i widoczne kolumny w perspektywach.

Tworzenie i używanie perspektyw

Do tworzenia perspektyw służy instrukcja `CREATE VIEW`, której uproszczona składnia ma postać:

```
CREATE [OR REPLACE] VIEW [{FORCE | NOFORCE}] VIEW nazwa_perspektywy
[(nazwa_aliasu [, nazwa_aliasu ...])] AS podzapytanie
[WITH {CHECK OPTION | READ ONLY} CONSTRAINT nazwa_więzów];
```

gdzie:

- OR REPLACE oznacza, że perspektywa zastępuje istniejącą.
- FORCE oznacza, że widok zostanie utworzony, nawet jeżeli tabele bazowe nie istnieją.
- NOFORCE oznacza, że widok nie zostanie utworzony, jeżeli tabele bazowe nie istnieją. Jest to ustawienie domyślne.
- *nazwa_perspektywy* jest nazwą perspektywy.
- *nazwa_aliasu* jest nazwą aliasu wyrażenia w podzapytaniu. Liczba aliasów musi być równa liczbie wyrażeń w podzapytaniu.
- *podzapytanie* jest podzapytaniem, które pobiera informacje z tabel bazowych. Jeżeli zostały podane aliasy, można je wykorzystać na liście po instrukcji `SELECT`.
- WITH CHECK OPTION oznacza, że wstawiane, modyfikowane lub usuwane będą jedynie takie wiersze, które mogą być pobrane przez perspektywę. Domyślnie wiersze nie są sprawdzane.
- WITH READ ONLY oznacza, że wiersze mogą być jedynie odczytywane z tabel podstawowych.
- *nazwa_więzów* jest nazwą więzów WITH CHECK OPTION lub WITH READ ONLY.

Wyróżniamy dwa główne typy perspektyw:

- proste perspektywy, które zawierają podzapytania pobierające informacje z jednej tabeli bazowej,
- perspektywy złożone, zawierające podzapytanie, które:
 - pobiera informacje z wielu tabel bazowych,
 - grupuje wiersze za pomocą klauzuli `GROUP BY` lub `DISTINCT`,
 - zawiera wywołanie funkcji.

W kolejnych podrozdziałach opisano, jak tworzyć tego typu perspektywy i używać ich.

Uprawnienia do tworzenia perspektyw

Do utworzenia perspektywy użytkownik musi posiadać uprawnienie `CREATE VIEW`. W poniższym przykładzie łączymy się jako użytkownik `system` i przyznajemy uprawnienie `CREATE VIEW` użytkownikowi `store`:

```
CONNECT system/oracle
GRANT CREATE VIEW TO store;
```

Tworzenie i używanie prostych perspektyw

Perspektywy proste korzystają z jednej tabeli bazowej. W poniższym przykładzie łączymy się jako użytkownik `store` i tworzymy perspektywę `cheap_products_view`, której podzapytanie pobiera wiersze zawierające niższą kwotę niż 15 zł:

```
CONNECT store/store_password
CREATE VIEW cheap_products_view AS
SELECT *
FROM products
WHERE price < 15;
```

W kolejnym przykładzie jest tworzona perspektywa `employees_view`, której podzapytanie pobiera z tabeli `employees` wszystkie kolumny oprócz `salary`:

```
CREATE VIEW employees_view AS
SELECT employee_id, manager_id, first_name, last_name, title
FROM employees;
```

Odpływanie perspektyw

Po utworzeniu perspektywy możemy jej użyć do uzyskania dostępu do tabeli. Poniższe zapytanie pobiera wiersze z perspektywy `cheap_products_view`:

```
SELECT product_id, name, price
FROM cheap_products_view;
```

PRODUCT_ID	NAME	PRICE
4	Wojny czołgów	13,95
6	2412: Powrót	14,95
7	Space Force 9	13,49
8	Z innej planety	12,99
9	Muzyka klasyczna	10,99
11	Twórczy wrzask	14,99
12	Pierwsza linia	13,49

Kolejny przykład pobiera wiersze z perspektywy `employees_view`:

```
SELECT *
FROM employees_view;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE
1	Jan	Kowalski	CEO	
2	1	Roman	Joświerz	Kierownik sprzedaży
3	2	Fryderyk	Helc	Sprzedawca
4	2	Zofia	Nowak	Sprzedawca

Wykonywanie instrukcji INSERT za pośrednictwem perspektywy

Korzystając z perspektywy `cheap_products_view`, możemy wykonywać instrukcje DML. W poniższym przykładzie za pomocą tej perspektywy jest wykonywana instrukcja `INSERT`, a następnie zostaje pobrany wstawiony wiersz:

```
INSERT INTO cheap_products_view (
    product_id, product_type_id, name, price
) VALUES (
    13, 1, 'Front zachodni', 13.50
);
```

1 row created.

```
SELECT product_id, name, price
FROM cheap_products_view
WHERE product_id = 13;
```

PRODUCT_ID	NAME	PRICE
13	Front zachodni	13,5



Instrukcje DML mogą być wykonywane jedynie z użyciem perspektyw prostych. Perspektywy złożone nie obsługują DML.

Ponieważ w perspektywie `cheap_products_view` nie jest używana opcja `WITH CHECK OPTION`, możemy wstawiać, modyfikować i usuwać wiersze, których perspektywa nie może pobrać. W poniższym przykła-

dzie jest wstawiany wiersz, w którym cena jest określona jako 16,50 zł (czyli więcej niż 15 zł, w związku z czym perspektywa nie może pobrać tego wiersza):

```
INSERT INTO cheap_products_view (
    product_id, product_type_id, name, price
) VALUES (
    14, 1, 'Front wschodni', 16.50
);
```

1 row created.

```
SELECT *
FROM cheap_products_view
WHERE product_id = 14;
```

no rows selected

Perspektywa employees_view zawiera podzapytanie, które z tabeli employees wybiera wszystkie kolumny oprócz salary. Jeżeli wykonamy instrukcję INSERT, używając tej perspektywy, w tabeli bazowej employees kolumna salary będzie miała wartość NULL:

```
INSERT INTO employees_view (
    employee_id, manager_id, first_name, last_name, title
) VALUES (
    5, 1, 'Jan', 'Kowal', 'CTO'
);
```

1 row created.

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE employee_id = 5;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
5	Jan	Kowal	

Kolumna salary ma wartość NULL.

Tworzenie perspektywy z więzami CHECK OPTION

Za pomocą więzów CHECK OPTION możemy określić, że instrukcje DML korzystające z perspektywy muszą spełniać podzapytanie. Na przykład poniższa instrukcja tworzy perspektywę cheap_products_view2 z więzami CHECK OPTION:

```
CREATE VIEW cheap_products_view2 AS
SELECT *
FROM products
WHERE price < 15
WITH CHECK OPTION CONSTRAINT cheap_products_view2_price;
```

W kolejnym przykładzie, korzystając z perspektywy cheap_products_view2, spróbujemy wstawić wiersz zawierający kwotę 19,50 zł. Baza danych zwraca błąd, ponieważ wiersz nie może zostać pobrany przez perspektywę:

```
INSERT INTO cheap_products_view2 (
    product_id, product_type_id, name, price
) VALUES (
    15, 1, 'Front południowy', 19.50
);
INSERT INTO cheap_products_view2 (
    *
);
ERROR at line 1:
ORA-01402: naruszenie klauzuli WHERE dla perspektywy z WITH CHECK OPTION
```

Tworzenie perspektywy z więzami READ ONLY

Dodając więcej READ ONLY, możemy utworzyć perspektywę tylko do odczytu. Na przykład poniższa instrukcja tworzy perspektywę cheap_products_view3 z więzami READ ONLY:

```
CREATE VIEW cheap_products_view3 AS
SELECT *
FROM products
WHERE price < 15
WITH READ ONLY CONSTRAINT cheap_products_view3_read_only;
```

W poniższym przykładzie spróbujemy wstawić wiersz za pośrednictwem perspektywy cheap_products_view3. Baza danych zwraca błąd, ponieważ perspektywa jest tylko do odczytu i nie zezwala na wykonywanie instrukcji DML:

```
INSERT INTO cheap_products_view3 (
    product_id, product_type_id, name, price
) VALUES (
    16, 1, 'Front północny', 19.50
);
product_id, product_type_id, name, price
*
ERROR at line 2:
ORA-42399: nie można przeprowadzić operacji DML na perspektywie tylko do odczytu
```

Uzyskiwanie informacji o definicjach perspektyw

Za pomocą polecenia DESCRIBE możemy uzyskać informacje o definicjach perspektyw. W poniższym przykładzie użyto go do pobrania informacji o perspektywie cheap_products_view3:

```
DESCRIBE cheap_products_view3
Name          Null?    Type
-----        -----
PRODUCT_ID      NOT NULL NUMBER(38)
PRODUCT_TYPE_ID      NUMBER(38)
NAME            NOT NULL VARCHAR2(30)
DESCRIPTION      VARCHAR2(50)
PRICE           NUMBER(5,2)
```

Informacje o perspektywach możemy również uzyskać z perspektywy user_views. W tabeli 11.9 opisano niektóre kolumny tej perspektywy.

Tabela 11.9. Wybrane kolumny perspektywy user_views

Kolumna	Typ	Opis
view_name	VARCHAR2(128)	Nazwa perspektywy
text_length	NUMBER	Liczba znaków w podzapytaniu perspektywy
text	LONG	Treść podzapytania perspektywy

W celu przejrzenia definicji perspektywy składowanej w kolumnie text musimy użyć polecenia SET LONG SQL*Plus, które ustawia liczbę znaków wyświetlana przez SQL*Plus przy pobieraniu kolumn typu LONG. Na przykład poniższe polecenie ustawia LONG na 200:

```
SET LONG 200
```

Poniższe zapytanie pobiera kolumny view_name, text_length i text z perspektywy user_views:

```
SELECT view_name, text_length, text
FROM user_views
ORDER BY view_name;
```

```
VIEW_NAME          TEXT_LENGTH
-----
TEXT
```

```
CHEAP_PRODUCTS_VIEW
```

```

SELECT "PRODUCT_ID","PRODUCT_TYPE_ID","NAME","DESCRIPTION","PRICE"
FROM products
WHERE price < 15

CHEAP_PRODUCTS_VIEW2          116
SELECT "PRODUCT_ID","PRODUCT_TYPE_ID","NAME","DESCRIPTION","PRICE"
FROM products
WHERE price < 15
WITH CHECK OPTION

CHEAP_PRODUCTS_VIEW3          113
SELECT "PRODUCT_ID","PRODUCT_TYPE_ID","NAME","DESCRIPTION","PRICE"
FROM products
WHERE price < 15
WITH READ ONLY

EMPLOYEES_VIEW                75
SELECT employee_id, manager_id, first_name, last_name, title
FROM employees

```

 **Uwaga** Informacje o wszystkich dostępnych perspektywach można uzyskać, odpytując perspektywę `all_views`.

Pobieranie informacji o więzach perspektywy

Wcześniej widzieliśmy, że do perspektywy możemy dodać więzy `CHECK OPTION` i `READ ONLY`. Perspektywa `cheap_products_view2` zawiera więzy `CHECK OPTION` dotyczące wysokości ceny (ma być niższa niż 15 zł). Perspektywa `cheap_products_view3` posiada więzy `READ ONLY` zapobiegające wprowadzaniu zmian do wierszy tabeli bazowej.

Informacje o więzach perspektywy możemy pobrać z perspektywy `user_constraints`, na przykład:

```

SELECT constraint_name, constraint_type, status, deferrable, deferred
FROM user_constraints
WHERE table_name IN ('CHEAP_PRODUCTS_VIEW2', 'CHEAP_PRODUCTS_VIEW3')
ORDER BY constraint_name;

```

CONSTRAINT_NAME	C	STATUS	DEFERRABLE	DEFERRED
CHEAP_PRODUCTS_VIEW2_PRICE	V	ENABLED	NOT DEFERRABLE	IMMEDIATE
CHEAP_PRODUCTS_VIEW3_READ_ONLY	O	ENABLED	NOT DEFERRABLE	IMMEDIATE

W kolumnie `constraint_type` dla więzów `CHEAP_PRODUCTS_VIEW2_PRICE` znajduje się wartość `V`, co jak widzieliśmy w tabeli 11.3, odpowiada więzom `CHECK OPTION`. W kolumnie `constraint_type` dla więzów `CHEAP_PRODUCTS_VIEW3_READ_ONLY` znajduje się wartość `O`, co odpowiada więzom `READ ONLY`.

Tworzenie i używanie perspektyw złożonych

Perspektywy złożone zawierają podzapytania, które:

- pobierają wiersze z wielu tabel,
- grupują wiersze za pomocą klauzuli `GROUP BY` lub `DISTINCT`,
- zawierają wywołanie funkcji.

W poniższym przykładzie jest tworzona perspektywa `products_and_types_view`, której podzapytanie wykonuje pełne złączenie zewnętrzne tabel `products` i `product_types` za pomocą składni SQL/92:

```

CREATE VIEW products_and_types_view AS
SELECT p.product_id, p.name product_name, pt.name product_type_name, p.price
FROM products p FULL OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.product_id;

```

Poniższy przykład odpytuje perspektywę `products_and_types_view`:

```
SELECT *
FROM products_and_types_view;
```

PRODUCT_ID	PRODUCT_NAME	PRODUCT_TY	PRICE
1	Nauka współczesna	Książka	19,95
2	Chemia	Książka	30
3	Supernowa	VHS	25,99
4	Wojny czołgów	VHS	13,95
5	Z Files	VHS	49,99
6	2412: Powrót	VHS	14,95
7	Space Force 9	DVD	13,49
8	Z innej planety	DVD	12,99
9	Muzyczka klasyczna	CD	10,99
10	Pop 3	CD	15,99
11	Twórczy wrzask	CD	14,99
12	Pierwsza linia		13,49
13	Front zachodni	Książka	13,5
14	Front wschodni	Książka	16,5
		Czasopismo	

W kolejnym przykładzie jest tworzona perspektywa `employee_salary_grades_view`, której podzapytanie wykorzystuje złączenie wewnętrzne do pobrania przedziałów płacowych pracowników:

```
CREATE VIEW employee_salary_grades_view AS
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e INNER JOIN salary_grades sg
ON e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY sg.salary_grade_id;
```

Poniższy przykład odpytuje perspektywę `employee_salary_grades_view`:

```
SELECT *
FROM employee_salary_grades_view;
```

FIRST_NAME	LAST_NAME	TITLE	SALARY	SALARY_GRADE_ID
Fryderyk	Hełc	Sprzedawca	150000	1
Zofia	Nowak	Sprzedawca	500000	2
Roman	Jóświerz	Kierownik sprzedaży	600000	3
Jan	Kowalski	CEO	800000	4

Kolejny przykład tworzy perspektywę `product_average_view`, której podzapytanie wykorzystuje:

- klauzulę WHERE do wybrania z tabeli `products` wierszy, w których kolumna `price` ma wartość mniejszą niż 15,
- klauzulę GROUP BY do pogrupowania wybranych wierszy według kolumny `product_type_id`,
- klauzulę HAVING do wybrania grup wierszy, w których średnia cena jest większa od 13 zł:

```
CREATE VIEW product_average_view AS
SELECT product_type_id, AVG(price) average_price
FROM products
WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY product_type_id;
```

Poniższy przykład odpytuje perspektywę `product_average_view`:

```
SELECT *
FROM product_average_view;
```

PRODUCT_TYPE_ID	AVERAGE_PRICE
1	13,5
2	14,45
3	13,24
	13,49

Modyfikowanie perspektywy

Z pomocą instrukcji CREATE OR REPLACE VIEW można zupełnie zastąpić perspektywę. W poniższym przykładzie użyto tej instrukcji do zastąpienia perspektywy product_average_view:

```
CREATE OR REPLACE VIEW product_average_view AS
SELECT product_type_id, AVG(price) average_price
FROM products
WHERE price < 12
GROUP BY product_type_id
HAVING AVG(price) > 11
ORDER BY product_type_id;
```

Z pomocą instrukcji ALTER VIEW możemy zmieniać więzy perspektywy. W poniższym przykładzie użyto tej instrukcji do usunięcia więzów cheap_products_view2_price z perspektywy cheap_products_view2:

```
ALTER VIEW cheap_products_view2
DROP CONSTRAINT cheap_products_view2_price;
```

Usuwanie perspektywy

Do usuwania perspektyw służy instrukcja DROP VIEW. W poniższym przykładzie usuwana jest perspektywa cheap_products_view2:

```
DROP VIEW cheap_products_view2;
```

Używanie niewidocznych kolumn w perspektywach

Nowością w Oracle Database 12c jest możliwość definiowania widocznych i niewidocznych kolumn w widoku. Za pomocą **VISIBLE** oznacza się kolumnę jako widoczną, a za pomocą **INVISIBLE** — jako niewidoczną. Jeśli nie określmy tego jawnie, kolumna domyślnie jest widoczna.

W poniższym przykładzie tworzona jest perspektywa employees_hidden_salary_view. Kolumna salary jest oznaczona jako INVISIBLE. Kolumna title jest jawnie oznaczona jako VISIBLE. Pozostałe kolumny są widoczne domyślnie.

```
CREATE VIEW employees_hidden_salary_view (
    employee_id, manager_id, first_name, last_name,
    title VISIBLE, salary INVISIBLE
) AS
SELECT employee_id, manager_id, first_name, last_name, title, salary
FROM employees;
```

Poniższe zapytanie pobiera wiersze z perspektywy. Można zauważyć, że kolumna salary nie jest wyświetlana w wynikach:

```
SELECT *
FROM employees_hidden_salary_view;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE
1		Jan	Kowalski	CEO
2	1	Roman	Józwierz	Kierownik sprzedaży
3	2	Fryderyk	Helc	Sprzedawca
4	2	Zofia	Nowak	Sprzedawca

Aby wyświetlić wśród wyników niewidoczną kolumnę, należy jawnie tego tego zażądać. W kolejnym przykładzie kolumna salary jest jawnie wymieniona na końcu listy kolumn do wyświetlenia:

```
SELECT employee_id, manager_id, first_name, last_name, title, salary
FROM employees_hidden_salary_view;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1		Jan	Kowalski	CEO	800000

2	1	Roman	Jóżwierz	Kierownik sprzedaży	600000
3	2	Fryderyk	Helc	Sprzedawca	150000
4	2	Zofia	Nowak	Sprzedawca	500000

Na tym zakończymy omówienie perspektyw. W kolejnym podrozdziale omówimy archiwa migawek.

Archiwa migawek

W archiwach migawek (ang. *Flashback Data Archives*), wprowadzonych w Oracle Database 11g, są składowane zmiany dokonane w tabeli w wyznaczonym okresie. Archiwa te dostarczają nam pełnego zapisu obserwacji. Po utworzeniu archiwum i dodaniu do niego tabel możemy:

- przeglądać wiersze w postaci, w jakiej były w określonym momencie (określonym przez datownik),
- przeglądać wiersze w postaci, w jakiej znajdowały się między dwoma datami.



W chwili pisania tego rozdziału tworzenie archiwów migawek nie jest możliwe w dołączanych bazach danych. Mowa tu o wersji 12.1.0.1.0 Oracle Database 12c.

Do tworzenia archiwów migawek służy instrukcja `CREATE FLASHBACK ARCHIVE`. W poniższym przykładzie łączymy się jako użytkownik `system` i tworzymy archiwum migawek o nazwie `test_archive`:

```
CONNECT system/manager
CREATE FLASHBACK ARCHIVE test_archive
TABLESPACE example
QUOTA 1 M
RETENTION 1 DAY;
```

Należy zauważyć, że:

- Archiwum jest tworzone w przestrzeni tabel `example`. Pełna lista przestrzeni tabel zostanie wyświetlona po uruchomieniu zapytania `SELECT tablespace_name FROM dba tablespaces`.
- Dla `test_archive` ustawiono limit rozmiaru wynoszący 1 megabajt, co oznacza, że archiwum może przechowywać maksymalnie 1 megabajt danych w przestrzeni tabel `example`.
- Dane w `test_archive` są przechowywane przez jeden dzień — po upływie tego czasu są usuwane.

Możemy zmienić istniejącą tabelę tak, aby składowała dane w archiwum:

```
ALTER TABLE store.products FLASHBACK ARCHIVE test_archive;
```

Od teraz wszystkie zmiany dokonywane w tabeli `store.products` będą zapisywane w archiwum. Poniższa instrukcja `INSERT` wstawia wiersz do tabeli `store.products`:

```
INSERT INTO store.products (
    product_id, product_type_id, name, description, price
) VALUES (
    15, 1, 'Jak korzystać z Linuksa?', 39.99
);
```

Poniższe zapytanie pobiera utworzony wiersz:

```
SELECT product_id, name, price
FROM store.products
WHERE product_id = 15;
```

PRODUCT_ID	NAME	PRICE
15	Jak korzystać z Linuksa?	39,99

Za pomocą poniższego zapytania możemy sprawdzić, jak wyglądały wiersze przed 5 minutami:

```
SELECT product_id, name, price
FROM store.products
AS OF TIMESTAMP
(SYSTIMESTAMP - INTERVAL '5' MINUTE);
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	19,95
2	Chemia	30
3	Supernowa	25,99
4	Wojny czołgów	13,95
5	Z Files	49,99
6	2412: Powrót	14,95
7	Space Force 9	13,49
8	Z innej planety	12,99
9	Muzyka klasyczna	10,99
10	Pop 3	15,99
11	Twórczy wrzask	14,99
12	Pierwsza linia	13,49
13	Front zachodni	13,5
14	Front wschodni	16,5

Należy zauważyć, że w wynikach zapytania brakuje nowego wiersza. Wynika to z tego, że został on dodany po dacie i godzinie określonej w zapytaniu (zakładamy, że przedstawiona wyżej instrukcja INSERT została uruchomiona mniej niż 5 minut temu).

Z pomocą poniższego zapytania możemy również przejrzeć wiersze w postaci, w jakiej znajdowały się w określonym dniu i godzinie (uruchamiając to zapytanie, należy zmienić datownik na datę i godzinę sprzed uruchomienia instrukcji INSERT):

```
SELECT product_id, name, price
FROM store.products
AS OF TIMESTAMP
TO_TIMESTAMP('2012-06-15 15:05:00', 'YYYY-MM-DD HH24:MI:SS');
```

Wyniki znowu nie będą zawierały nowego wiersza, ponieważ został on wstawiony do tabeli po dacie i godzinie określonych w zapytaniu.

Z pomocą poniższego zapytania możemy przejrzeć wiersze w postaci, w jakiej znajdowały się między dwiema datami (ponownie należy zmienić datownik):

```
SELECT product_id, name, price
FROM store.products VERSIONS BETWEEN TIMESTAMP
TO_TIMESTAMP('2012-06-15 12:00:00', 'YYYY-MM-DD HH24:MI:SS')
AND TO_TIMESTAMP('2012-06-15 15:59:59', 'YYYY-MM-DD HH24:MI:SS');
```

Z pomocą poniższego zapytania możemy przejrzeć wiersze w postaci, w jakiej znajdowały się między określona datą a obecną (ponownie należy zmienić datownik):

```
SELECT product_id, name, price
FROM store.products VERSIONS BETWEEN TIMESTAMP
TO_TIMESTAMP('2012-06-15 13:45:52', 'YYYY-MM-DD HH24:MI:SS')
AND MAXVALUE;
```

Z pomocą instrukcji ALTER TABLE możemy również zatrzymać archiwizowanie danych tabeli, na przykład:

```
ALTER TABLE store.products NO FLASHBACK ARCHIVE;
```

Tworząc tabelę, możemy przypisać jej archiwum migawek, na przykład:

```
CREATE TABLE store.test_table (
  id INTEGER,
  name VARCHAR2(10)
) FLASHBACK ARCHIVE test_archive;
```

Informacje o archiwach są dostępne w następujących perspektywach:

- user_flashback_archive i dba_flashback_archive — wyświetlają ogólne informacje o archiwach,
- user_flashback_archive_ts i dba_flashback_archive_ts — zawierają informacje o przestrzeniach tabel, w których znajdują się archiwa,
- user_flashback_archive_tables i dba_flashback_archive_tables — wyświetlają informacje o archiwizowanych tabelach.

Archiwa migawek można zmieniać. Na przykład poniższa instrukcja wydłuża okres przechowywania danych do dwóch lat:

```
ALTER FLASHBACK ARCHIVE test_archive  
MODIFY RETENTION 2 YEAR;
```

Możliwe jest również usunięcie danych z archiwum przed upływem okresu trwałości danych. Na przykład poniższa instrukcja usuwa dane starsze niż jeden dzień:

```
ALTER FLASHBACK ARCHIVE test_archive  
PURGE BEFORE TIMESTAMP(SYSTIMESTAMP - INTERVAL '1' DAY);
```

Można również usunąć wszystkie dane z archiwum:

```
ALTER FLASHBACK ARCHIVE test_archive PURGE ALL;
```

Archiwum migawek można również usunąć, na przykład:

```
DROP FLASHBACK ARCHIVE test_archive;
```

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- tabele są tworzone za pomocą instrukcji CREATE TABLE,
- sekwencje generują sekwencję liczb całkowitych,
- indeks bazy danych może przyspieszyć uzyskiwanie dostępu do wierszy,
- widok jest predefiniowanym zapytaniem jednej lub wielu tabel,
- w archiwum migawek są składowane zmiany dokonane w tabeli w wyznaczonym okresie.

W następnym rozdziale zajmiemy się programowaniem w PL/SQL.

ROZDZIAŁ

12

Wprowadzenie do programowania w PL/SQL

Bazy danych Oracle zawierają proceduralny język programowania PL/SQL (ang. *Procedural Language/SQL*). Umożliwia on pisanie programów zawierających instrukcje SQL.

W tym rozdziale zostały opisane następujące zagadnienia związane z PL/SQL:

- struktury blokowe,
- zmienne i typy,
- logika warunkowa,
- pętle,
- kursory umożliwiające PL/SQL odczytywanie wyników zwróconych przez zapytanie,
- wyjątki wykorzystywane do obsługi błędów,
- procedury,
- funkcje,
- pakiety wykorzystywane do grupowania procedur i funkcji w jednostki,
- wyzwalacze będące blokami kodu wykonywanymi, gdy w bazie danych wystąpi określone zdarzenie,

Korzystając z PL/SQL, możemy dodać logikę biznesową do aplikacji bazodanowej. Taka skoncentrowana logika biznesowa może być wykorzystywana przez dowolny program uzyskujący dostęp do bazy danych, na przykład SQL*Plus, programy napisane w Javie, C# itp.



Należy ponownie uruchomić skrypt *store_schema.sql* w celu przywrócenia pierwotnej postaci tabel, aby wyniki zapytań były zgodne z prezentowanymi w tym rozdziale.

Blokи

Programy pisane w PL/SQL są podzielone na struktury zwane **blokami**, z których każdy zawiera instrukcje PL/SQL i SQL. Blok PL/SQL ma następującą strukturę:

```
[DECLARE  
  instrukcje_deklarujace  
]  
BEGIN  
  instrukcje_wykonywane
```

```
[EXCEPTION
  instrukcje_obsługujace_wyjątki
]
END;
/
```

gdzie:

- *instrukcje_deklarującce* deklarują zmienne używane w pozostałej części bloku PL/SQL. Bloki DECLARE są opcjonalne.
- *instrukcje_wykonywane* są faktycznymi instrukcjami wykonywanymi — mogą to być pętle, logika warunkowa itd.
- *instrukcje_obsługujące_wyjątki* są instrukcjami obsługującymi błędy wykonywania, które mogą wystąpić po uruchomieniu bloku. Bloki EXCEPTION są opcjonalne.

Blok PL/SQL jest kończony ukośnikiem (/).

Poniższy kod (znajdujący się w skrypcie *area_example.sql* w katalogu SQL) oblicza szerokość prostokąta przy danym polu i wysokości:

```
SET SERVEROUTPUT ON

DECLARE
  v_szerokosc      INTEGER;
  v_wysokosc        INTEGER := 2;
  v_powierzchnia   INTEGER := 6;
BEGIN
  -- ustaw szerokość na wartość równą polu powierzchni podzielonemu przez wysokość
  v_szerokosc := v_powierzchnia / v_wysokosc;
  DBMS_OUTPUT.PUT_LINE('v_szerokosc = ' || v_szerokosc);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Dzielenie przez zero');
END;
/
```

Polecenie SET SERVEROUTPUT ON włącza wyjście serwera, dzięki czemu po uruchomieniu skryptu w PL/SQL wiersze generowane przez DBMS_OUTPUT.PUT_LINE() będą wyświetlane na ekranie. Po tym początkowym poleceniu rozpoczyna się faktyczny blok PL/SQL, podzielony na bloki DECLARE, BEGIN i EXCEPTION.

Blok DECLARE zawiera deklaracje trzech zmiennych typu INTEGER o nazwach *v_szerokosc*, *v_wysokosc* i *v_powierzchnia*. Zmienne *v_wysokosc* i *v_powierzchnia* są inicjalizowane z wartościami 2 i 6.

W bloku BEGIN są trzy wiersze. Pierwszy z nich jest komentarzem zawierającym tekst „ustaw szerokość na wartość równą polu powierzchni podzielonemu przez wysokość”. Drugi wiersz ustawia wartość zmiennej *v_szerokosc* jako iloraz wartości zmiennej *v_powierzchnia* i *v_wysokosc*; to oznacza, że zmiennej *v_szerokosc* jest przypisywana wartość 3 (=6/2). Trzeci wiersz wywołuje DBMS_OUTPUT.PUT_LINE() w celu wyświetlenia wartości zmiennej *v_szerokosc* na ekranie. DBMS_OUTPUT jest wbudowanym pakietem kodu dostarczanym z Oracle Database. Zawiera między innymi procedury umożliwiające wypisywanie wartości na ekranie.

Blok EXCEPTIONS obsługuje wszelkie próby wykonania dzielenia przez zero. W tym celu „wyłapywany” jest wyjątek ZERO_DIVIDE. W tym przykładzie nie występuje dzielenie przez zero, jeżeli jednak zmienimy wartość *v_height* na zero i uruchomimy skrypt, wystąpi wyjątek.

Na samym końcu skryptu ukośnik (/) oznacza koniec bloku PL/SQL.

Wynik uruchomienia skryptu *area_example.sql* w SQL*Plus:

```
SQL> @ C:\SQL\area_example.sql
v_width = 3
```



Jeżeli plik *area_example.sql* znajduje się w innym katalogu niż C:\SQL, należy odpowiednio zmienić powyższe polecenie. W systemach Unix i Linux należy użyć zwykłych ukośników.

Zmienne i typy

Zmienne są deklarowane wewnątrz bloku **DECLARE**. Jak można było zauważyć we wcześniejszym przykładzie, deklaracja zmiennej zawiera zarówno jej nazwę, jak i typ. Na przykład zmienna **v_width** była deklarowana jako:

```
v_width INTEGER;
```

 **Uwaga** Typy w PL/SQL są podobne do typów kolumn bazy danych. Wszystkie zostały opisane w dodatku.

Poniżej przedstawiono więcej przykładów deklaracji (te zmienne mogą być użyte do przechowywania wartości kolumn z tabeli **products**):

```
v_product_id      INTEGER;
v_product_type_id INTEGER;
v_name            VARCHAR2(30);
v_description     VARCHAR2(50);
v_price           NUMBER(5, 2);
```

Typ zmiennej możemy również zdefiniować za pomocą słowa kluczowego **%TYPE**. Wówczas zostanie użyty typ przypisany określonej kolumnie. W poniższym przykładzie użyto słowa kluczowego **%TYPE** do zadeklarowania zmiennej tego samego typu co kolumna **price** tabeli **products**, czyli **NUMBER(5, 2)**:

```
v_product_price product.price%TYPE
```

Logika warunkowa

Do wykonywania logiki warunkowej służą słowa kluczowe **IF**, **THEN**, **ELSE**, **ELSIF** i **END IF**:

```
IF warunek1 THEN
  instrukcje1
ELSIF warunek2 THEN
  instrukcje2
ELSE
  instrukcje3
END IF;
```

gdzie:

- *warunek1* i *warunek2* są wyrażeniami boolowskimi szacowanymi jako prawda lub fałsz,
- *instrukcje1*, *instrukcje2* i *instrukcje3* są instrukcjami PL/SQL.

Przepływ logiki warunkowej jest następujący:

- jeżeli *warunek1* jest spełniony, wykonywane są *instrukcje1*,
- jeżeli *warunek1* jest fałszywy, ale spełniony jest *warunek2*, wykonywane są *instrukcje2*,
- jeżeli nie jest spełniony ani *warunek1*, ani *warunek2*, wykonywane są *instrukcje3*.

Instrukcję **IF** można również osadzić w innej instrukcji **IF**, co obrazuje poniższy przykład:

```
IF v_count > 0 THEN
  v_message := 'v_count jest liczbą dodatnią';
  IF v_area > 0 THEN
    v_message := 'v_count i v_area są liczbami dodatnimi';
  END IF
ELSIF v_count = 0 THEN
  v_message := 'v_count ma wartość zero';
ELSE
  v_message := 'v_count jest liczbą ujemną';
END IF;
```

Jeżeli w powyższym przykładzie `v_count` jest większa od zera, zmiennej `v_message` jest przypisywana wartość '`v_count` jest liczbą dodatnią'. Jeżeli wartości zmiennych `v_count` i `v_area` są większe od zera, zmiennej `v_message` jest przypisywana wartość '`v_count` i `v_area` są liczbami dodatnimi'. Reszta jest łatwa do zrozumienia.

Pętle

Pętle służą do uruchamiania instrukcji zero lub więcej razy. W PL/SQL dostępne są trzy rodzaje pętli:

- **proste pętle** wykonywane aż do jawnego zakończenia pętli,
- **pętle WHILE** wykonywane aż do zastąpienia określonego warunku,
- **pętle FOR** wykonywane określona liczbę razy.

Wszystkie rodzaje pętli zostaną opisane w kolejnych podrozdziałach.

Proste pętle

Prosta pętla jest wykonywana aż do jej jawnego zakończenia. Jej składnia ma następującą postać:

```
LOOP
  instrukcje
END LOOP;
```

Do zakończenia pętli służy instrukcja `EXIT` lub `EXIT WHEN`. Instrukcja `EXIT` natychmiast przerywa pętlę, instrukcja `EXIT WHEN` przerzuca natomiast pętlę przy wystąpieniu określonego warunku.

Poniższy przykład prezentuje prostą pętlę. Przed rozpoczęciem pętli zmienna `v_counter` jest inicjalizowana z wartością 0. Pętla dodaje 1 do wartości `v_counter` i jest przerywana za pomocą instrukcji `EXIT WHEN`, gdy wartość `v_counter` wyniesie 5:

```
v_counter := 0;
LOOP
  v_counter := v_counter + 1;
  EXIT WHEN v_counter = 5;
END LOOP;
```



Instrukcję `EXIT WHEN` można umieścić w dowolnym miejscu pętli.

W Oracle Database 11g i nowszych można również zakończyć bieżącą iterację pętli, korzystając z instrukcji `CONTINUE` lub `CONTINUE WHEN`. Instrukcja `CONTINUE` kończy bieżącą iterację bezwarunkowo i przechodzi do następnej iteracji. Instrukcja `CONTINUE WHEN` kończy bieżącą iterację pętli, jeżeli wystąpi określony warunek, i następnie przechodzi do kolejnej iteracji. Poniższy przykład obrazuje użycie instrukcji `CONTINUE`:

```
v_counter := 0;
LOOP
  -- po wykonaniu instrukcji CONTINUE sterowanie powróci do tego punktu
  v_counter := v_counter + 1;
  IF v_counter = 3 THEN
    CONTINUE; -- bezwarunkowe zakończenie bieżącej iteracji
  END IF;
  EXIT WHEN v_counter = 5;
END LOOP;
```

Kolejny przykład obrazuje użycie instrukcji `CONTINUE WHEN`:

```
v_counter := 0;
LOOP
  -- po wykonaniu instrukcji CONTINUE WHEN sterowanie powróci do tego punktu
  v_counter = v_counter + 1;
```

```
CONTINUE WHEN v_counter = 3; -- kończy bieżącą iterację, jeżeli v_counter = 3
EXIT WHEN v_counter = 5;
END LOOP;
```



Instrukcje CONTINUE i CONTINUE WHEN nie mogą przekroczyć granicy procedury, funkcji lub metody.

Pętle WHILE

Pętla WHILE jest wykonywana aż do wystąpienia określonego warunku. Jej składnia ma następującą postać:

```
WHILE warunek LOOP
  instrukcje
END LOOP;
```

Poniższy przykład przedstawia pętlę WHILE wykonywaną, dopóki wartość zmiennej v_counter jest mniejsza niż 6:

```
v_counter := 0;
WHILE v_counter < 6 LOOP
  v_counter := v_counter + 1;
END LOOP;
```

Pętle FOR

Pętla FOR jest wykonywana określoną liczbę razy. Liczbę iteracji pętli określamy za pomocą **dolnej i górnej granicy** wartości zmiennej pętli. Zmienna pętli jest wówczas zwiększana (lub zmniejszana) przy każdej iteracji. Składnia pętli FOR ma następującą postać:

```
FOR zmienna_pętli IN [REVERSE] dolna_granica..górnna_granica LOOP
  instrukcje
END LOOP;
```

gdzie:

- *zmienna_pętli* jest zmienną pętli. Można użyć już istniejącej zmiennej lub pozwolić pętli na utworzenie nowej (dzieje się tak, jeżeli podana przez nas zmienna nie istnieje). Wartość zmiennej pętli jest powiększana (lub zmniejszana, jeżeli zostanie użyte słowo kluczowe REVERSE) o 1 przy każdej iteracji.
- REVERSE oznacza, że wartość zmiennej pętli powinna być dekrementowana przy każdej iteracji. Zmienna pętli jest inicjalizowana z górną granicą i jej wartość jest zmniejszana o 1 aż do osiągnięcia dolnej granicy. Dolną granicę należy określić przed górną.
- *dolina_granica* jest dolną granicą. Zmienna pętli jest inicjalizowana z tą wartością, jeżeli nie zostanie użyte słowo kluczowe REVERSE.
- *górnna_granica* jest górną granicą. Jeżeli zostanie użyte słowo kluczowe REVERSE, zmienna pętli zostanie zainicjalizowana z tą wartością.

Poniższy przykład przedstawia pętlę FOR. Należy zauważyć, że zmienna v_counter2 nie jest jawnie deklarowana — pętla FOR automatycznie tworzy nową zmienną v_counter2 o typie INTEGER:

```
FOR v_counter IN 1..5 LOOP
  DBMS_OUTPUT.PUT_LINE(v_counter2);
END LOOP;
```

W poniższym przykładzie użyto słowa kluczowego REVERSE:

```
FOR v_counter IN REVERSE 1..5 LOOP
  DBMS_OUTPUT.PUT_LINE(v_counter2);
END LOOP;
```

W tym przykładzie zmienna v_counter2 ma początkowo wartość 5 i przy każdej iteracji pętli jest zmniejszana o 1.

Kursory

Kursor służy do pobierania wierszy zwróconych przez zapytanie. Wiersze są pobierane do kursora za pomocą zapytania, a następnie są kolejno odczytywane z niego. Korzystanie z kursorów zwykle przebiega w pięciu krokach:

1. Deklarowanie zmiennych, w których będą przechowywane wartości kolumn dla wiersza.
2. Deklarowanie kursora. Deklaracja zawiera zapytanie.
3. Otwarcie kursora.
4. Pobieranie kolejnych wierszy z kursora i zapisywanie wartości kolumn w zmiennych zadeklarowanych w punkcie 1. Następnie robimy coś z tymi zmiennymi, na przykład wyświetlamy ich wartości na ekranie, używamy w obliczeniach itd.
5. Zamknięcie kursora.

Kroki te zostaną szczegółowo opisane w kolejnych podrozdziałach. Zostanie również przedstawiony prosty przykład pobierający kolumny `product_id`, `name` i `price` z tabeli `products`.

Krok 1. — deklarowanie zmiennych przechowujących wartości kolumn

Pierwszy etap to zadeklarowanie zmiennych, w których będą przechowywane wartości kolumn. Te zmienne muszą być kompatybilne z typami kolumn.

 **Wskazówka** Widzieliśmy wcześniej, że do pobrania typu kolumny służy słowo kluczowe `%TYPE`. Jeżeli użyjemy go przy deklarowaniu typu zmiennych, automatycznie uzyskają one odpowiedni typ.

W poniższym przykładzie są deklarowane trzy zmienne, w których będą przechowywane wartości kolumn `product_id`, `name` i `price` tabeli `products`. Należy zauważyć, że użyto słowa kluczowego `%TYPE`, aby typ każdej zmiennej został automatycznie ustawiony na typ odpowiedniej kolumny:

```
DECLARE
  v_product_id products.product_id%TYPE;
  v_name       products.name%TYPE;
  v_price      products.price%TYPE;
```

Krok 2. — deklaracja kursora

Drugi etap to deklaracja kursora. Składa się ona z nazwy, którą przypisujemy kursorowi, oraz zapytania, jakie chcemy uruchomić. Deklaracja kursora, jak wszystkie inne deklaracje, jest umieszczana w sekcji deklaracji. Składnia deklaracji kursora ma następującą postać:

```
CURSOR nazwa_kursora IS
  instrukcja_SELECT;
```

gdzie:

- *nazwa_kursora* jest nazwą kursora,
- *instrukcja_SELECT* jest zapytaniem.

W poniższym przykładzie jest deklarowany kursor o nazwie `v_product_cursor`, którego zapytanie pobiera kolumny `product_id`, `name` i `price` z tabeli `products`:

```
CURSOR v_product_cursor IS
  SELECT product_id, name, price
  FROM products
  ORDER BY product_id;
```

Zapytanie nie jest uruchamiane aż do otwarcia kursora.

Krok 3. — otwarcie kursora

Trzeci etap to otwarcie kursora. Służy do tego instrukcja OPEN, która musi zostać umieszczona w wykonywalnej sekcji bloku.

W poniższym przykładzie otwierany jest kursor v_product_cursor, co powoduje wykonanie zapytania:

```
OPEN v_product_cursor;
```

Krok 4. — pobieranie wierszy z kursora

Czwarty etap to pobieranie wierszy z kursora, do czego służy instrukcja FETCH. Odczytuje ona wartości kolumn do zmiennych zadeklarowanych w punkcie 1. Składnia tej instrukcji ma następującą postać:

```
FETCH nazwa_kursora
INTO zmieniona[, zmieniona ...];
```

gdzie:

- *nazwa_kursora* jest nazwą kursora,
- *zmieniona* jest nazwą zmiennej, w której będzie przechowywana wartość kolumny z kursora. Dla każdej wartości kolumny należy zapewnić odpowiednią zmienną.

W poniższym przykładzie instrukcja FETCH pobiera wiersz z kursora v_product_cursor i zapisuje wartości kolumn do zmiennych v_product_id, v_name i v_price utworzonych w kroku 1.:

```
FETCH v_product_cursor
INTO v_product_id, v_name, v_price;
```

Ponieważ kursor może zawierać wiele wierszy, do ich odczytania jest potrzebna pętla. Do określenia warunku jej przerwania możemy użyć zmiennej boolowskiej v_product_cursor%NOTFOUND. Ma ona wartość true, jeżeli z kursora v_product_cursor zostały odczytane wszystkie wiersze. Poniżej przedstawiono kod tej pętli:

```
LOOP
    -- pobieranie wierszy z kursora
    FETCH v_product_cursor
    INTO v_product_id, v_name, v_price;

    -- przerwij pętlę, gdy nie ma więcej wierszy, na co wskazuje
    -- zmenna boolowska v_product_cursor%NOTFOUND (=true, jeżeli nie ma więcej wierszy)
    EXIT WHEN v_product_cursor%NOTFOUND;

    -- użyj DBMS_OUTPUT.PUT_LINE() do wyświetlenia zmiennych
    DBMS_OUTPUT.PUT_LINE(
        'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
        ', v_price = ' || v_price
    );
END LOOP;
```

Należy zauważyć, że użycie DBMS_OUTPUT.PUT_LINE() do wyświetlania wartości zmiennych v_product_id, v_name i v_price odczytanych dla każdego wiersza. W prawdziwej aplikacji zmienna v_price mogłaby zostać użyta w złożonych obliczeniach.

Krok 5. — zamknięcie kursora

Piąty etap to zamknięcie kursora za pomocą instrukcji CLOSE. Zwalnia ono zasoby systemowe. Poniższy przykład zamyka kursor v_product_cursor:

```
CLOSE v_product_cursor;
```

W kolejnym podrozdziale został przedstawiony pełny skrypt zawierający wszystkie pięć etapów.

Pełny przykład — product_cursor.sql

Poniższy skrypt *product_cursor.sql* znajduje się w katalogu *SQL*:

```
-- product_cursor.sql wyświetla, korzystając z kurSORA, wartości kolumn
-- product_id, name i price tabeli products
```

```
SET SERVEROUTPUT ON
```

```
DECLARE
    -- krok 1 deklaracja zmiennych
    v_product_id products.product_id%TYPE;
    v_name        products.name%TYPE;
    v_price       products.price%TYPE;

    -- krok 2 deklaracja kurSORA
    CURSOR v_product_cursor IS
        SELECT product_id, name, price
        FROM products
        ORDER BY product_id;
BEGIN
    -- krok 3 otwarcie kurSORA
    OPEN v_product_cursor;

    LOOP
        -- krok 4 pobieranie wierszy z kurSORA
        FETCH v_product_cursor
        INTO v_product_id, v_name, v_price;

        -- przerwij pętlę, gdy nie ma więcej wierszy, na co wskazuje
        -- zmienna boolowska v_product_cursor%NOTFOUND (=true, jeżeli nie ma więcej wierszy)
        EXIT WHEN v_product_cursor%NOTFOUND;

        -- użyj DBMS_OUTPUT.PUT_LINE() do wyświetlenia zmiennych
        DBMS_OUTPUT.PUT_LINE(
            'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
            ', v_price = ' || v_price
        );
    END LOOP;

    -- krok 5 zamknięcie kurSORA
    CLOSE v_product_cursor;
END;
/
```

W celu uruchomienia tego skryptu należy:

1. połączyć się z bazą danych jako użytkownik **store**, podając hasło **store_password**,
2. uruchomić skrypt *product_cursor.sql* za pomocą SQL*Plus:

```
SQL> @ C:\SQL\product_cursor.sql
```

Wyjście skryptu *product_cursor.sql* ma następującą postać:

```
v_product_id = 1, v_name = Nauka współczesna, v_price = 19,95
v_product_id = 2, v_name = Chemia, v_price = 30
v_product_id = 3, v_name = Supernowa, v_price = 25,99
v_product_id = 4, v_name = Wojny czolgów, v_price = 13,95
v_product_id = 5, v_name = Z Files, v_price = 49,99
v_product_id = 6, v_name = 2412: Powrót, v_price = 14,95
v_product_id = 7, v_name = Space Force 9, v_price = 13,49
v_product_id = 8, v_name = Z innej planety, v_price = 12,99
v_product_id = 9, v_name = Muzyka klasyczna, v_price = 10,99
v_product_id = 10, v_name = Pop 3, v_price = 15,99
v_product_id = 11, v_name = Twórczy wrzask, v_price = 14,99
v_product_id = 12, v_name = Pierwsza linia, v_price = 13,49
```

Kursory i pętle FOR

Dostęp do wierszy kursora możemy uzyskać za pomocą pętli FOR. W takim przypadku nie jest konieczne jawne otwieranie kursora, ponieważ pętla czyni to automatycznie. W poniższym skrypcie *product_cursor2.sql* użyto pętli FOR do pobrania wierszy z kursora *v_product_cursor*. Należy zauważyć, że ten skrypt jest krótszy niż *product_cursor.sql*:

```
-- product_cursor2.sql wyświetla wartości kolumn product_id, name,
-- i price tabeli product, korzystając z kursora i pętli FOR
```

```
SET SERVEROUTPUT ON

DECLARE
  CURSOR v_product_cursor IS
    SELECT product_id, name, price
    FROM products
    ORDER BY product_id;
BEGIN
  FOR v_product IN v_product_cursor LOOP
    DBMS_OUTPUT.PUT_LINE(
      'product_id = ' || v_product.product_id ||
      ', name = ' || v_product.name ||
      ', price = ' || v_product.price
    );
  END LOOP;
END;
/
```

W celu uruchomienia skryptu *product_cursor2.sql* należy użyć następującego polecenia:

```
SQL> @ "C:\SQL\product_cursor2.sql"
```

Wyjście z tego skryptu ma następującą postać:

```
product_id = 1, name = Nauka współczesna, price = 19,95
product_id = 2, name = Chemia, price = 30
product_id = 3, name = Supernowa, price = 25,99
product_id = 4, name = Wojny czołgów, price = 13,95
product_id = 5, name = Z Files, price = 49,99
product_id = 6, name = 2412: Powrót, price = 14,95
product_id = 7, name = Space Force 9, price = 13,49
product_id = 8, name = Z innej planety, price = 12,99
product_id = 9, name = Muzyka klasyczna, price = 10,99
product_id = 10, name = Pop 3, price = 15,99
product_id = 11, name = Twórczy wrzask, price = 14,99
product_id = 12, name = Pierwsza linia, price = 13,49
```

Instrukcja OPEN-FOR

Z kursorem można również użyć instrukcji OPEN-FOR, dającej większą swobodę dzięki temu, że umożliwia ona przypisanie kursora innemu zapytaniu. Obrazuje to skrypt *product_cursor3.sql*:

```
-- product_cursor3.sql wyświetla wartości kolumn product_id, name,
-- i price tabeli product, korzystając ze zmiennej kursora oraz instrukcji OPEN-FOR
```

```
SET SERVEROUTPUT ON

DECLARE
  -- deklaracja typu REF CURSOR o nazwie t_product_cursor
  TYPE t_product_cursor IS
    REF CURSOR RETURN products%ROWTYPE;

  -- deklaracja obiektu t_product_cursor o nazwie v_product_cursor
  v_product_cursor t_product_cursor;
```

```
-- deklaracja obiektu v_product (typu products%ROWTYPE),
-- w którym będą przechowywane wartości kolumn tabeli products
v_product products%ROWTYPE;
BEGIN
  -- przypisanie zapytania do v_product_cursor
  -- i otwarcie kurSORA za pomocą instrukcji OPEN-FOR
  OPEN v_product_cursor FOR
    SELECT * FROM products WHERE product_id < 5;

  -- pętla pobierająca wiersze z v_product_cursor do v_product
  LOOP
    FETCH v_product_cursor INTO v_product;
    EXIT WHEN v_product_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(
      'product_id = ' || v_product.product_id ||
      ', name = ' || v_product.name ||
      ', price = ' || v_product.price
    );
  END LOOP;

  -- zamknięcie kurSORa v_product_cursor
  CLOSE v_product_cursor;
END;
/
```

W bloku DECLARE poniższa instrukcja deklaruje typ REF CURSOR o nazwie t_product_cursor:

```
TYPE t_product_cursor IS
REF CURSOR RETURN products%ROWTYPE
```

REF CURSOR jest wskaźnikiem do kurSORa i przypomina wskaźnik z języka programowania C++. Poniższa instrukcja deklaruje typ t_product_cursor zdefiniowany przez użytkownika i zwraca wiersz zawierający różne kolumny tabeli products (wskaże na to słowo kluczowe %ROWTYPE). Taki typ zdefiniowany przez użytkownika może być użyty do zadeklarowania faktycznego obiektu, co obrazuje poniższa instrukcja, w której jest deklarowany obiekt o nazwie v_product_cursor:

```
v_product_cursor t_product_cursor;
```

Poniższa instrukcja deklaruje obiekt o nazwie v_product (typu products%ROWTYPE), w którym będą przechowywane kolumny tabeli products:

```
v_product products%ROWTYPE
```

W bloku BEGIN kurSORowi v_product_cursor jest przypisywane zapytanie i jest on otwierany przez następującą instrukcję OPEN-FOR:

```
OPEN v_product_cursor FOR
SELECT * FROM products WHERE product_id < 5;
```

Po wykonaniu tej instrukcji do kurSORa v_product_cursor zostaną załadowane pierwsze cztery wiersze tabeli products. KurSORowi v_product_cursor można przypisać każdą prawidłową instrukcję SELECT, co oznacza, że kurSOR może zostać użyty ponownie z innym zapytaniem.

Następnie poniższa pętla pobiera wiersze z v_product_cursor do v_product i wyświetla poszczególne wartości:

```
LOOP
  FETCH v_product_cursor INTO v_product;
  EXIT WHEN v_product_cursor%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(
    'product_id = ' || v_product.product_id ||
    ', name = ' || v_product.name ||
    ', price = ' || v_product.price
  );
END LOOP;
```

Po zakończeniu pętli kurSOR jest zamknięty za pomocą następującej instrukcji:

```
CLOSE v_product_cursor;
```

W celu uruchomienia skryptu *product_cursor3.sql* należy użyć następującego polecenia:

```
SQL> @ "C:\SQL\product_cursor3.sql"
```

Wyjście z tego skryptu ma następującą postać:

```
product_id = 1, name = Nauka współczesna, price = 19,95
product_id = 2, name = Chemia, price = 30
product_id = 3, name = Supernowa, price = 25,99
product_id = 4, name = Wojny czołgów, price = 13,95
```

Kursory bez ograniczenia

Kursory opisane w poprzednim podrozdziale miały określony typ zwracanych danych. Tego typu kursory nazywamy **kursorami z ograniczeniem**. Typ zwracany w ich przypadku musi być zgodny z kolumnami w zapytaniu uruchamianym przez kurSOR. **KurSOR bez ograniczenia** nie ma zdefiniowanego typu zwarcanego, może więc uruchamiać dowolne zapytanie.

Sposób użycia kurSORa bez ograniczenia został przedstawiony w poniższym skrypcie *unconstrained_cursor.sql*. KurSOR *v_cursor* został użyty do uruchomienia dwóch zapytań:

-- Ten skrypt pokazuje użycie kurSORów bez ograniczenia

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
-- deklaracja typu REF CURSOR o nazwie t_cursor (nie ma on zwracanego typu,
-- więc może uruchomić dowolne zapytanie)
TYPE t_cursor IS REF CURSOR;
```

```
-- deklaracja obiektu v_cursor typu t_cursor
v_cursor t_cursor;
```

```
-- deklaracja obiektu v_product (typu products%ROWTYPE),
-- w którym będą przechowywane kolumny z tabeli products
v_product products%ROWTYPE;
```

```
-- deklaracja obiektu v_customer (typu customers%ROWTYPE),
-- w którym będą przechowywane kolumny z tabeli customers
v_customer customers%ROWTYPE;
```

```
BEGIN
```

```
-- przypisanie zapytania kurSORowi v_cursor i otwarcie go za pomocą OPEN FOR
OPEN v_cursor FOR
SELECT * FROM products WHERE product_id < 5;
```

```
-- użycie pętli do pobrania wierszy z v_cursor do v_product
LOOP
```

```
  FETCH v_cursor INTO v_product;
  EXIT WHEN v_cursor%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(
    'product_id = ' || v_product.product_id ||
    ', name = ' || v_product.name ||
    ', price = ' || v_product.price
  );
END LOOP;
```

```
-- przypisanie nowego zapytania kurSORowi v_cursor i otwarcie go za pomocą OPEN FOR
OPEN v_cursor FOR
SELECT * FROM customers WHERE customer_id < 3;
```

```
-- użycie pętli do pobrania wierszy z v_cursor do v_customer
LOOP
```

```
  FETCH v_cursor INTO v_customer;
  EXIT WHEN v_cursor%NOTFOUND;
```

```

DBMS_OUTPUT.PUT_LINE(
  'customer_id = ' || v_customer.customer_id ||
  ', first_name = ' || v_customer.first_name ||
  ', last_name = ' || v_customer.last_name
);
END LOOP;

-- zamknięcie v_cursor
CLOSE v_cursor;
END;
/

```

W celu uruchomienia skryptu *unconstrained_cursor.sql* należy użyć następującego polecenia:

```
SQL> @ "C:\SQL\unconstrained_cursor.sql"
```

Wyjście z tego skryptu ma następującą postać:

```

product_id = 1, name = Nauka współczesna, price = 19,95
product_id = 2, name = Chemia, price = 30
product_id = 3, name = Supernowa, price = 25,99
product_id = 4, name = Wojny czołgów, price = 13,95
customer_id = 1, first_name = Jan, last_name = Nikiel
customer_id = 2, first_name = Lidia, last_name = Stal

```

Zmienne REF CURSOR zostaną opisane dokładniej w tym rozdziale, a typy definiowane przez użytkownika — w kolejnym.

Wyjątki

Wyjątki służą do obsługi błędów powstających w trakcie wykonywania kodu PL/SQL. Wcześniej zaprezentowano poniższy przykład PL/SQL zawierający blok EXCEPTION:

```

DECLARE
  v_width  INTEGER;
  v_height INTEGER := 2;
  v_area   INTEGER := 6;
BEGIN
  -- szerokość ma być równa ilorazowi pola i wysokości
  v_width := v_area / v_height;
  DBMS_OUTPUT.PUT_LINE('v_width = ' || v_width);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Dzielenie przez zero');
END;
/

```

Blok EXCEPTION w tym przykładzie obsługuje dzielenie liczby przez zero. W terminologii PL/SQL *przechwytyuje* on wyjątek ZERO_DIVIDE zgłoszany w bloku BEGIN (choć w prezentowanym kodzie wyjątek ZERO_DIVIDE nigdy nie jest zgłoszany). Wyjątek ZERO_DIVIDE i inne często używane zostały opisane w tabeli 12.1.

Tabela 12.1. Predefiniowane wyjątki

Wyjątek	Błąd	Opis
ACCESS_INTO_NULL	ORA-06530	Próba przypisania wartości atrybutom niezainicjalizowanego obiektu. (Obiekty zostaną opisane w rozdziale 13.)
CASE_NOT_FOUND	ORA-06592	Żadna z klauzul WHEN instrukcji CASE nie została wybrana, a nie zdefiniowano domyślnej klauzuli ELSE
COLLECTION_IS_NULL	ORA-06531	Próba wywołania metody (innej niż EXIST) kolekcji na niezainicjalizowanej, zagnieżdżonej tablicy lub VARRAY albo próba przypisania wartości elementom niezainicjalizowanej, zagnieżdżonej tablicy lub VARRAY. (Kolekcje zostaną opisane w rozdziale 14.)

Tabela 12.1. Predefiniowane wyjątki — ciąg dalszy

Wyjątek	Błąd	Opis
CURSOR_ALREADY_OPEN	ORA-06511	Próba otwarcia już otwartego kursora. Kursor musi zostać zamknięty przed ponownym otwarciem
DUP_VAL_ON_INDEX	ORA-00001	Próba zapisania powtarzających się wartości w kolumnie ograniczonej przez unikatowy indeks
INVALID_CURSOR	ORA-01001	Próba wykonania nieprawidłowej operacji na kursorze, na przykład zamknięcie nieotwartego kursora
INVALID_NUMBER	ORA-01722	Próba konwersji napisu znakowego na liczbę nie powiodła się, ponieważ nie reprezentuje on prawidłowej liczby W PL/SQL zamiast INVALID_NUMBER jest zgłaszane VALUE_ERROR
LOGIN_DENIED	ORA-01017	Próba połączenia z bazą danych z użyciem nieprawidłowej nazwy użytkownika lub hasła
NO_DATA_FOUND	ORA-01403	Instrukcja SELECT INTO nie zwróciła żadnych wierszy lub spróbowało uzyskać dostęp do usuniętego elementu w tabeli zagnieżdżonej bądź niezainicjalizowanego elementu w tabeli indeksu
NOT_LOGGED_IN	ORA-01012	Próba uzyskania dostępu do elementu bazy danych przy nienawiązanym połączeniu z bazą
PROGRAM_ERROR	ORA-06501	PL/SQL napotkał problem wewnętrzny
ROWTYPE_MISMATCH	ORA-06504	Zmienna kursora macierzystego i zmienna kursora PL/SQL biorące udział w przypisaniu mają niekompatybilne typy zwracania Na przykład jeżeli otwarta zmienna kursora macierzystego jest przesyłana do zapisanej procedury lub funkcji, typy zwracane rzeczywistych i formalnych parametrów muszą być zgodne
SELF_IS_NULL	ORA-30625	Próba wywołania metody MEMBER obiektu NULL. Co oznacza, że wbudowany parametr SELF (który jest zawsze pierwszym parametrem przesyłanym do metody MEMBER) wynosi NULL
STORAGE_ERROR	ORA-06500	Modułowi PL/SQL zabrakło pamięci lub została ona uszkodzona
SUBSCRIPT_BEYOND_COUNT	ORA-065333	Próba odwołania się do elementu zagnieżdżonej tabeli lub VARRAY za pomocą większej wartości indeksu niż liczba elementów w kolekcji
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Próba odwołania się do elementu zagnieżdżonej tabeli lub VARRAY za pomocą liczby indeksu spoza prawidłowego zakresu (na przykład -1)
SYS_INVALID_ROWID	ORA-01410	Konwersja napisu znakowego na uniwersalny ROWID nie powiodła się, ponieważ napis znakowy nie reprezentuje prawidłowego ROWID
TIMEOUT_ON_RESOURCE	ORA-00051	Upłynął limit czasu oczekiwania na zasób przez bazę danych
TOO_MANY_ROWS	ORA-01422	Instrukcja SELECT INTO zwróciła więcej niż jeden wiersz
VALUE_ERROR	ORA-06502	Wystąpił błąd podczas obliczania, konwersji, przycinania lub związany z rozmiarami Na przykład jeżeli podczas wybierania wartości kolumny do zmiennej znakowej wartość jest dłuższa od zadeklarowanej długości zmiennej, PL/SQL przerwa przypisywanie i zgłasza VALUE_ERROR Uwaga: W instrukcjach PL/SQL wyjątek VALUE_ERROR jest zgłoszany, jeżeli konwersja napisu znakowego na liczbę zakończy się niepowodzeniem. W instrukcjach SQL jest zgłoszony wyjątek INVALID_VALUE
ZERO_DIVIDE	ORA-01476	Próba podzielenia liczby przez zero

W kolejnych podrozdziałach zostaną przedstawione przykłady niektórych wyjątków wymienionych w tabeli 12.1.

Wyjątek ZERO_DIVIDE

Wyjątek `ZERO_DIVIDE` jest zgłoszany, gdy wystąpi próba podzielenia liczby przez zero. W poniższym przykładzie w bloku `BEGIN` występuje podzielenie 1 przez 0, co powoduje zgłoszenie wyjątku `ZERO_DIVIDE`:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(1 / 0);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Dzielenie przez zero');
END;
/
```

Dzielenie przez zero

Po zgłoszeniu wyjątku sterowanie jest przekazywane do bloku `EXCEPTION` i w klauzulach `WHEN` są wyszukiwane odpowiednie wyjątki. Następnie jest wykonywany kod z tej klauzuli. W powyższym przykładzie w bloku `BEGIN` jest zgłoszany wyjątek `ZERO_DIVIDE` i sterowanie jest przekazywane do bloku `EXCEPTION`. Zgodny wyjątek zostaje odnaleziony i jest wykonywany kod z odpowiedniej klauzuli `WHEN`.

Jeżeli nie zostanie odnaleziony odpowiedni wyjątek, jest on przekazywany do ogólniejszego bloku. Gdybyśmy na przykład pominęli blok `EXCEPTION` w powyższym przykładzie, wyjątek zostałby przekazany do SQL*Plus:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(1 / 0);
END;
/
BEGIN
*
BŁĄD w linii 1:
ORA-01476: dzielnik jest równy zero
ORA-06512: przy linia 2
```

Jak widać, SQL*Plus wyświetli domyślny komunikat o błędzie, zawierający numery wierszy, kod błędu Oracle oraz krótki opis.

Wyjątek DUP_VAL_ON_INDEX

Wyjątek `DUP_VAL_ON_INDEX` jest zgłoszany, jeżeli nastąpi próba zapisu powtarzającej się wartości do kolumny z ograniczeniem indeksu unikatowego.

W poniższym przykładzie próbujemy wstawić do tabeli `customers` wiersz z `customer_id` równym 1. To powoduje zgłoszenie wyjątku `DUP_VAL_ON_INDEX`, ponieważ tabela `customers` zawiera już wiersz z `customer_id` wynoszącym 1:

```
BEGIN
  INSERT INTO customers (
    customer_id, first_name, last_name
  ) VALUES (
    1, 'Grzegorz', 'Zielony'
  );
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Powtarzająca się wartość indeksu.');
END;
/
```

Powtarzająca się wartość indeksu.

Wyjątek INVALID_NUMBER

Wyjątek `INVALID_NUMBER` jest zgłoszany, gdy zostanie dokonana próba konwersji nieprawidłowego napisu znakowego na liczbę. W poniższym przykładzie próbujemy przekonwertować napis 123X na liczbę

użyta w instrukcji INSERT, co powoduje zgłoszenie wyjątku INVALID_NUMBER, ponieważ 123X nie jest prawidłową liczbą:

```
BEGIN
  INSERT INTO customers (
    customer_id, first_name, last_name
  ) VALUES (
    '123X', 'Grzegorz', 'Zielony'
  );
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Konwersja napisu na liczbę nie powiodła się.');
END;
/

```

Konwersja napisu na liczbę nie powiodła się.

Wyjątek OTHERS

Wyjątek OTHERS może posłużyć do obsługi wszystkich zgłaszanych wyjątków:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(1 / 0);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Wystąpił wyjątek.');
END;
/

```

Wystąpił wyjątek.

Ponieważ OTHERS obsługuje wszystkie wyjątki, w bloku EXCEPTION przypadek ten musi zostać umieszczony za wszystkimi wymienianymi wyjątkami. Jeżeli umieścimy OTHERS w innym miejscu, zostanie zwrócony komunikat o błędzie PLS-00370, na przykład:

```
SQL> BEGIN
  2  DBMS_OUTPUT.PUT_LINE(1 / 0);
  3  EXCEPTION
  4  WHEN OTHERS THEN
  5    DBMS_OUTPUT.PUT_LINE('Wystąpił wyjątek.');
  6  WHEN ZERO_DIVIDE THEN
  7    DBMS_OUTPUT.PUT_LINE('Dzielenie przez zero.');
  8 END;
  9 /
WHEN OTHERS THEN
*
BŁĄD w linii 4:
ORA-06550: linia 4, kolumna 3:
PLS-00370: procedura OTHERS musi być ostatnia w ciągu definiowanych procedur obsługujących wyjątków dla bloku
ORA-06550: linia 0, kolumna 0:
PL/SQL: Compilation unit analysis terminated
```

Procedury

Procedura zawiera grupę instrukcji SQL i PL/SQL. Procedury umożliwiają centralizowanie logiki biznesowej w bazie danych i mogą być używane przez każdy program uzyskujący dostęp do bazy danych.

Z tego podrozdziału dowiesz się, jak:

- utworzyć procedurę,
- wywołać procedurę,
- uzyskać informacje o procedurach,

- usunąć procedurę,
- znaleźć błędy w procedurze.

Tworzenie procedury

Do tworzenia procedur służy instrukcja `CREATE PROCEDURE`. Jej uproszczona składnia ma następującą postać:

```
CREATE [OR REPLACE] PROCEDURE nazwa_procedury
[nazwa_parametru [IN | OUT | IN OUT] typ [, ...]]
{IS | AS}
BEGIN
treść_procedury
END nazwa_procedury;
```

gdzie:

- OR REPLACE oznacza, że procedura ma zastąpić istniejącą.
- *nazwa_procedury* jest nazwą procedury.
- *nazwa_parametru* jest nazwą parametru przesyłanego do procedury. Procedura może przyjąć wiele parametrów.
- IN | OUT | IN OUT jest **trybem** parametru. Można wybrać następujące tryby parametrów:
 - IN jest domyślnym trybem parametru i oznacza, że parametr musi mieć wartość w momencie uruchomienia procedury. Wartość parametru IN nie może być zmieniona w treści procedury.
 - OUT oznacza, że parametr jest ustawiany w treści procedury.
 - IN OUT oznacza, że parametr może mieć wartość w momencie uruchamiania procedury i może ona zostać zmieniona w treści procedury.
- *typ* jest typem parametru.
- *treść_procedury* zawiera kod procedury.

W poniższym przykładzie jest tworzona procedura `update_product_price()`. Ta procedura oraz pozostały kod PL/SQL, prezentowany w dalszej części tego rozdziału, zostały utworzone przez skrypt `store_schema.sql`. Mnoży ona cenę produktu razy czynnik. Jako parametry są do niej przesyłane identyfikator produktu oraz mnożnik. Jeżeli produkt istnieje, procedura mnoży jego cenę razy czynnik i zatwierdza zmianę:

```
CREATE PROCEDURE update_product_price(
    p_product_id IN products.product_id%TYPE,
    p_factor      IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- zlicza produkty z przesłanym
    -- product_id (powinno być 1, jeżeli produkt istnieje)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- jeżeli produkt istnieje (v_product_count = 1),
    -- aktualizuje jego cene
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
```

```

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END update_product_price;
/

```

Ta procedura przyjmuje dwa parametry o nazwach `p_product_id` i `p_factor`. Obydwa wykorzystują tryb IN, co oznacza, że ich wartości muszą być ustawione w momencie uruchomienia procedury i nie mogą być zmienione w jej treści.

Sekcja z deklaracjami zawiera zmienną `v_product_count` typu INTEGER:

```
v_product_count INTEGER
```

Treść procedury zaczyna się za słowem kluczowym BEGIN. Instrukcja SELECT w treści procedury pobiera z tabeli products liczbę wierszy, w których `product_id` jest równe `p_product_id`:

```

SELECT COUNT(*)
  INTO v_product_count
  FROM products
 WHERE product_id = p_product_id;

```



Instrukcja COUNT(*) zwraca liczbę znalezionych wierszy.

Jeżeli produkt zostanie odnaleziony, zmienna `v_product_count` będzie miała wartość 1; w przeciwnym razie będzie miała wartość 0. Jeżeli `v_product_count` ma wartość 1, wartość z kolumny `price` jest mnożona razy `p_factor` za pomocą instrukcji UPDATE, a następnie zmiana jest zatwierdzana:

```

IF v_product_count = 1 THEN
  UPDATE products
    SET price = price * p_factor
   WHERE product_id = p_product_id;
  COMMIT;
END IF;

```

Blok EXCEPTION wykonuje ROLLBACK, jeżeli zgłoszony zostanie jakikolwiek wyjątek:

```

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;

```

Słowo kluczowe END oznacza koniec procedury:

```

END update_product_price;
/

```



Powtórzenie nazwy procedury po słowie kluczowym END nie jest obowiązkowe, stanowi jednak dobrą praktykę.

Wywoływanie procedury

Do **wywoływania** procedur służy instrukcja CALL. W przykładzie prezentowanym w tym podrozdziale cena produktu nr 1 jest mnożona razy 1,5 za pomocą procedury utworzonej w poprzednim podrozdziale. Poniższe zapytanie pobiera najpierw cenę produktu nr 1, abyśmy mogli ją porównać z ceną zmodyfikowaną:

```

SELECT price
  FROM products
 WHERE product_id = 1;

```

```

PRICE
-----
19,95

```

Poniższa instrukcja wywołuje procedurę `update_product_price()`, przesyłając do niej parametry o wartościach 1 (`product_id`) oraz 1,5 (mnożnik ceny produktu):

```
CALL update_product_price(1, 1.5);
```

Call completed.

Ta instrukcja stanowi przykład zastosowania **zapisu pozycyjnego** do przesyłania wartości do procedury lub funkcji. W zapisie pozycyjnym wartości są przypisywane na podstawie pozycji parametrów. W powyższym przykładzie pierwszą wartością w wywołaniu jest 1 i ta wartość jest przesyłana do pierwszego parametru w procedurze (`p_product_id`). Drugą wartością w wywołaniu jest 1.5 (1,5) i ta wartość jest przesyłana do drugiego parametru (`p_factor`).

Kolejne zapytanie ponownie pobiera informacje o produkcie nr 1. Należy zauważyć, że cena została pomnożona razy 1,5:

```
SELECT price
FROM products
WHERE product_id = 1;
```

```
PRICE
-----
29,93
```

W Oracle Database 11g i nowszych można przesyłać parametry, używając zapisu nazywanego i mieszanego. W **zapisie nazywanym** dodajemy nazwę parametru przy wywoływaniu procedury. Na przykład poniższa instrukcja wywołuje procedurę `update_product_price()` z użyciem zapisu nazywanego. Należy zauważyć, że wartości parametrów `p_factor` i `p_product_id` są wskazywane za pomocą `=>`:

```
CALL update_product_price(p_factor => 1.3, p_product_id => 2);
```

 **Zapis nazywany** ułatwia czytanie kodu, a także jego konserwację, ponieważ parametry są jawnie wymienione.

W **zapisie mieszanym** jest stosowany zarówno zapis pozycyjny, jak i nazywany. Zapis pozycyjny jest używany dla pierwszego zestawu parametrów, a nazwany — dla ostatniego zestawu. Zapis mieszały jest przydatny, gdy pracujemy z procedurami lub funkcjami, które mają zarówno parametry obowiązkowe, jak i opcjonalne. Wówczas dla parametrów obowiązkowych używamy zapisu pozycyjnego, a dla parametrów opcjonalnych — nazywanego. W powyższym przykładzie użyto zapisu mieszanego. Należy zauważyć, że przy określaniu wartości parametrów zapis pozycyjny występuje przed zapisem nazywanym:

```
CALL update_product_price(3, p_factor => 1.7);
```

Uzyskiwanie informacji o procedurach

Informacje o procedurach możemy pobrać z perspektywy `user_procedures`. W tabeli 12.2 opisano wybrane kolumny perspektywy `user_procedures`.

Tabela 12.2. Wybrane kolumny perspektywy `user_procedures`

Kolumna	Typ	Opis
OBJECT_NAME	VARCHAR2(30)	Nazwa obiektu, którym może być procedura, funkcja lub pakiet
PROCEDURE_NAME	VARCHAR2(30)	Nazwa procedury
AGGREGATE	VARCHAR2(3)	Określa, czy procedura jest funkcją agregującą (YES lub NO)
IMPLTYPEOWNER	VARCHAR2(30)	Właściciel typu (jeśli występuje)
IMPLTYPEOWNER	VARCHAR2(30)	Nazwa typu (jeśli występuje)
PARALLEL	VARCHAR2(3)	Określa, czy procedura jest włączona dla zapytań równoległych (YES lub NO)

W powyższym przykładzie z perspektywy `user_procedures` pobierane są wartości kolumn `object_name`, `aggregate` i `parallel` dla procedury `update_product_price()`:

```
SELECT object_name, aggregate, parallel
FROM user_procedures
```

```
WHERE object_name = 'UPDATE_PRODUCT_PRICE';
```

OBJECT_NAME	AGG PAR
UPDATE_PRODUCT_PRICE	NO NO

 **Uwaga** Informacje o wszystkich dostępnych procedurach można pobrać z perspektywy `all_procedures`.

Usuwanie procedury

Do usuwania procedur służy instrukcja `DROP PROCEDURE`. Na przykład poniższa instrukcja usuwa procedurę `update_product_price()`:

```
DROP PROCEDURE update_product_price;
```

Przeglądanie błędów w procedurze

Jeżeli baza danych zgłosi błąd przy tworzeniu procedury, możemy przejrzeć błędy za pomocą polecenia `SHOW ERRORS`. Na przykład poniższa instrukcja `CREATE PROCEDURE` próbuje utworzyć procedurę, w której w szóstym wierszu występuje błąd składni (prawidłowa nazwa parametru to `p_dob`, a nie `p_dobs`):

```
SQL> CREATE PROCEDURE update_customer_dob (
  2   p_customer_id INTEGER, p_dob DATE
  3 ) AS
  4 BEGIN
  5   UPDATE customers
  6   SET dob = p_dobs
  7   WHERE customer_id = p_customer_id;
  8 END update_customer_dob;
  9 /
```

Warning: Procedure created with compilation errors.

Jak widzimy, wystąpił błąd komplikacji. Listę błędów możemy wyświetlić za pomocą polecenia `SHOW ERRORS`:

```
SQL> SHOW ERRORS
Błędy dla PROCEDURE UPDATE_CUSTOMER_DOB:
```

LINE/COL ERROR

```
-----  
5/3      PL/SQL: SQL Statement ignored  
6/13      PL/SQL: ORA-00904: "P_DOBS": niepoprawny identyfikator
```

Piąty wiersz został pominięty, ponieważ w szóstym występuje odwołanie do nieprawidłowej nazwy kolumny. Możemy naprawić ten błąd za pomocą polecenia `EDIT` dla instrukcji `CREATE PROCEDURE` — należy zmienić `p_dobs` na `p_dob` i ponownie uruchomić instrukcję za pomocą skrótu `/`.

Funkcje

Funkcja przypomina procedurę z tą różnicą, że musi zwracać wartość. Zapisane procedury i funkcje są czasem nazywane **składowanymi podprogramami**, ponieważ stanowią one małe programy.

Z tego podrozdziału dowiesz się, jak:

- utworzyć funkcję,
- wywołać ją,
- uzyskać informacje o funkcjach,
- usunąć funkcję.

Tworzenie funkcji

Do tworzenia funkcji służy instrukcja `CREATE FUNCTION`. Jej uproszczona składnia ma następującą postać:

```
CREATE [OR REPLACE] FUNCTION nazwa_funkcji
[nazwa_parametru [IN | OUT | IN OUT] typ [, ...])
RETURN typ
{IS | AS}
BEGIN
treść_funkcji
END nazwa_funkcji;
```

gdzie:

- `OR REPLACE` oznacza, że funkcja ma zastąpić istniejącą funkcję.
- `nazwa_funkcji` jest nazwą funkcji.
- `nazwa_parametru` jest nazwą parametru przesyłanego do funkcji. Do funkcji można przesyłać kilka parametrów.
- `IN | OUT | IN OUT` oznacza tryb parametru.
- `typ` jest typem parametru.
- `treść_funkcji` zawiera kod funkcji. W przeciwieństwie do procedury funkcja musi zwracać wartość o typie określonym w klauzuli `RETURN`.

W poniższym przykładzie jest tworzona funkcja o nazwie `circle_area()`, która zwraca pole koła. Promień koła jest przesyłany jako parametr o nazwie `p_radius`:

```
CREATE FUNCTION circle_area (
    p_radius IN NUMBER
) RETURN NUMBER AS
    v_pi    NUMBER := 3.1415926;
    v_area NUMBER;
BEGIN
    -- pole koła jest obliczane jako iloczyn liczby PI i kwadratu promienia
    v_area := v_pi * POWER(p_radius, 2);
    RETURN v_area;
END circle_area;
/
```

W kolejnym przykładzie jest tworzona funkcja `average_product_price()`, zwracająca średnią cenę produktów, których `product_type_id` jest równy przesłanej wartości parametru:

```
CREATE FUNCTION average_product_price (
    p_product_type_id IN INTEGER
) RETURN NUMBER AS
    v_average_product_price NUMBER;
BEGIN
    SELECT AVG(price)
    INTO v_average_product_price
    FROM products
    WHERE product_type_id = p_product_type_id;
    RETURN v_average_product_price;
END average_product_price;
/
```

Wywoływanie funkcji

Utworzone funkcje wywołujemy tak jak funkcje wbudowane do bazy danych, co zostało opisane w rozdziale 4. Dla przypomnienia: funkcję możemy wywołać za pomocą instrukcji `SELECT`, umieszczając w klauzuli `FROM` tabelę `dual`. W poniższym przykładzie jest wywoływana funkcja `circle_area()` dla promienia wynoszącego 2 (zapis pozycyjny):

```
SELECT circle_area(2)
FROM dual;
```

```
CIRCLE_AREA(2)
-----
12,5663704
```

W Oracle Database 11g i nowszych, wywołując funkcję, można również zastosować zapis nazywany lub mieszany. Na przykład w poniższym zapytaniu użyto zapisu nazywanego przy wywoływaniu funkcji `circle_area()`:

```
SELECT circle_area(p_radius => 4)
FROM dual;
```

```
CIRCLE_AREA(P_RADIUS=>4)
-----
50,2654816
```

W kolejnym przykładzie jest wywoływana funkcja `average_product_price()`, do której jest przesyłana wartość parametru 1 w celu uzyskania średniej ceny produktów o `product_type_id` wynoszącym 1:

```
SELECT average_product_price(1)
FROM dual;
```

```
AVERAGE_PRODUCT_PRICE(1)
-----
29,965
```

Uzyskiwanie informacji o funkcjach

Informacje o funkcjach można pobrać z perspektywy `user_procedures`. Została ona już opisana w podrozdziale „Uzyskiwanie informacji o procedurach”. W poniższym przykładzie z perspektywy `user_procedures` są pobierane kolumny `object_name`, `aggregate` i `parallel` dla funkcji `circle_area()` i `average_product_price()`:

```
SELECT object_name, aggregate, parallel
FROM user_procedures
WHERE object_name IN ('CIRCLE_AREA', 'AVERAGE_PRODUCT_PRICE');
```

OBJECT_NAME	AGG	PAR
CIRCLE_AREA	NO	NO
AVERAGE_PRODUCT_PRICE	NO	NO

Usuwanie funkcji

Do usuwania funkcji służy instrukcja `DROP FUNCTION`. Na przykład poniższa instrukcja usuwa funkcję `circle_area()`:

```
DROP FUNCTION circle_area;
```

Pakiety

Z tego podrozdziału dowiesz się, w jaki sposób można grupować procedury i funkcje w **pakietach**. Pakiety umożliwiają hermetyzację powiązanych funkcjonalności w niezależną jednostkę. Modularyzacja kodu PL/SQL z użyciem pakietów umożliwia tworzenie bibliotek kodu, które mogą być wykorzystywane przez innych programistów.

Baza danych Oracle jest dostarczana z biblioteką pakietów, które umożliwiają między innymi uzyskiwanie dostępu do plików zewnętrznych, zarządzanie bazą danych, generowanie kodu HTML. Wszystkie pakiety zostały opisane w podręczniku *Oracle Database Packages and Types Reference* wydanym przez Oracle Corporation.

Pakiety składają się zwykle ze **specyfikacji** i **treści**. W specyfikacji pakietu są wymienione wszystkie dostępne procedury, funkcje, typy i obiekty. Elementy te mogą być udostępnione wszystkim użytkownikom bazy danych — czasem nazywa się je **publicznymi** (choć są dostępne jedynie dla użytkowników mających uprawnienia dostępu do pakietu). Specyfikacja nie zawiera kodu składającego się na procedury i funkcje — kod znajduje się w treści pakietu.

Wszystkie elementy w treści, które nie zostały wymienione w specyfikacji, są **prywatknymi** elementami pakietu. Łącząc elementy publiczne i prywatne, możemy utworzyć pakiet, którego złożoność jest ukryta przed użytkownikami. Jest to jeden z głównych celów programowania — ukrywanie złożoności przed użytkownikami.

Tworzenie specyfikacji pakietu

Do tworzenia specyfikacji pakietu służy instrukcja `CREATE PACKAGE`. Jej uproszczona składnia ma następującą postać:

```
CREATE [OR REPLACE] PACKAGE nazwa_pakietu
{IS | AS}
specyfikacja_pakietu
END nazwa_pakietu;
```

gdzie:

- *nazwa_pakietu* jest nazwą pakietu,
- *specyfikacja_pakietu* jest listą publicznych procedur, funkcji, typów i obiektów, dostępnych dla użytkowników pakietu.

Poniższy przykład tworzy specyfikację pakietu o nazwie `product_package`:

```
CREATE PACKAGE product_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_products_ref_cursor RETURN t_ref_cursor;
  PROCEDURE update_product_price (
    p_product_id IN products.product_id%TYPE,
    p_factor      IN NUMBER
  );
END product_package;
/
```

Typ `t_ref_cursor` jest typem `REF CURSOR` w PL/SQL. Typ `REF CURSOR` przypomina wskaźnik z języka C++ i wskazuje na kursor. Jak już wiesz, kursor umożliwia odczytywanie wierszy zwróconych przez zapytanie.

Funkcja `get_products_ref_cursor()` zwraca `t_ref_cursor` i jak przekonasz się w następnym podroziale, wskazuje na kursor zawierający wiersze pobrane z tabeli `products`.

Procedura `update_product_price()` mnoży cenę produktu i zatwierdza zmianę.

Tworzenie treści pakietu

Do tworzenia treści pakietu służy instrukcja `CREATE PACKAGE BODY`. Jej uproszczona składnia to:

```
CREATE [OR REPLACE] PACKAGE BODY nazwa_pakietu
{IS | AS}
treść_pakietu
END nazwa_pakietu;
```

gdzie:

- *nazwa_pakietu* jest nazwą pakietu zgodną z nazwą pakietu w specyfikacji,
- *treść_pakietu* zawiera kod procedur i funkcji.

Poniższy przykład tworzy treść pakietu `product_package`:

```
CREATE PACKAGE BODY product_package AS
  FUNCTION get_products_ref_cursor
```

```

RETURN t_ref_cursor IS
    products_ref_cursor t_ref_cursor;
BEGIN
    -- pobierz REF CURSOR
    OPEN products_ref_cursor FOR
        SELECT product_id, name, price
        FROM products;
    -- zwraca REF CURSOR
    RETURN products_ref_cursor;
END get_products_ref_cursor;

PROCEDURE update_product_price (
    p_product_id IN products.product_id%TYPE,
    p_factor      IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- zlicza produkty z przesłanym
    -- product_id (jeżeli produkt istnieje, powinno być 1)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- jeżeli produkt istnieje (v_product_count = 1),
    -- aktualizuje jego cenę
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
END product_package;
/

```

Funkcja `get_products_ref_cursor()` otwiera kursor i pobiera kolumny `product_id`, `name` i `price` z tabeli `products`. Odwołanie do tego kursora (`REF CURSOR`) jest zapisywane w `v_products_ref_cursor` i zwarcane przez funkcję.

Procedura `update_product_price()` mnoży cenę produktu i zatwierdza zmianę. Jest ona taka sama jak procedura przedstawiona w podrozdziale „Tworzenie procedury”, więc nie będziemy jej szczegółowo omawiali.

Wywoływanie funkcji i procedur z pakietu

Wywołując funkcje i procedury z pakietu, należy zamieścić jego nazwę w wywołaniu. W poniższym przykładzie jest wywoływana funkcja `product_package.get_products_ref_cursor()`, zwracająca odwołanie do kursora zawierającego `product_id`, `name` i `price` produktów:

```

SELECT product_package.get_products_ref_cursor
FROM dual;

GET_PRODUCTS_REF_CUR
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

PRODUCT_ID NAME          PRICE
-----  -----

```

1 Nauka współczesna	19,95
2 Chemia	30
3 Supernowa	25,99
4 Wojny czołgów	13,95
5 Z Files	49,99
6 2412: Powrót	14,95
7 Space Force 9	13,49
8 Z innej planety	12,99
9 Muzyka klasyczna	10,99
10 Pop 3	15,99
11 Twórczy wrzask	14,99
12 Pierwsza linia	13,49

W kolejnym przykładzie jest wywoływana funkcja `product_package.update_product_price()` w celu pomnożenia ceny produktu nr 3 razy 1,25:

```
CALL product_package.update_product_price(3, 1.25);
```

Poniższe zapytanie pobiera informacje o produkcie nr 3. Należy zauważyć, że cena została zwiększoną:

```
SELECT price
FROM products
WHERE product_id = 3;
```

PRICE

32,49

Uzyskiwanie informacji o funkcjach i procedurach w pakiecie

Informacje o funkcjach i procedurach z pakietu możemy uzyskać z perspektywy `user_procedures`. Została ona opisana w podrozdziale „Uzyskiwanie informacji o procedurach”. Poniższy przykład pobiera z perspektywy `user_procedures` kolumny `object_name` i `procedure_name` dla pakietu `product_package`:

```
SELECT object_name, procedure_name
FROM user_procedures
WHERE object_name = 'PRODUCT_PACKAGE';
```

OBJECT_NAME	PROCEDURE_NAME
-----	-----
PRODUCT_PACKAGE	GET_PRODUCTS_REF_CURSOR
PRODUCT_PACKAGE	UPDATE_PRODUCT_PRICE

Usuwanie pakietu

Do usuwania pakietów służy instrukcja `DROP PACKAGE`. Na przykład poniższa instrukcja usuwa pakiet `product_package`:

```
DROP PACKAGE product_package;
```

Wyzwalacze

Wyzwalacz jest procedurą uruchamianą automatycznie przez bazę danych, gdy dla danej tabeli zostanie uruchomiona określona instrukcja DML (`INSERT`, `UPDATE` lub `DELETE`). Wyzwalacze są przydatne na przykład przy wykonywaniu zaawansowanych inspekcji zmian wprowadzonych do wartości kolumn w tabeli.

Kiedy uruchamiany jest wyzwalacz

Wyzwalacz może być uruchamiany przed uruchomieniem instrukcji DML. Ponadto, ponieważ instrukcja DML może mieć wpływ na więcej niż jeden wiersz, kod wyzwalacza może być uruchamiany raz dla każdego z tych wierszy (**wyzwalacz poziomu wierszy**) lub tylko raz dla wszystkich wierszy (**wyzwalacz poziomu instrukcji**).

Na przykład jeżeli utworzymy wyzwalacz poziomu wierszy dla instrukcji UPDATE tabeli i uruchomimy instrukcję UPDATE modyfikującą 10 wierszy, wyzwalacz zostanie uruchomiony 10 razy. Jeżeli jednak utworzymy wyzwalacz poziomu instrukcji, zostanie on uruchomiony raz dla całej instrukcji UPDATE niezależnie od liczby wierszy, na które ma ona wpływ.

Między wyzwalaczami poziomu wierszy i instrukcji występuje jeszcze jedna różnica. Wyzwalacz poziomu wierszy ma dostęp do starych i nowych wartości kolumny, jeżeli zostanie on uruchomiony w następstwie instrukcji UPDATE dotyczącej tej kolumny. Uruchomienie wyzwalacza poziomu wierszy może być również ograniczone za pomocą **warunku** wyzwalacza. Na przykład można utworzyć warunek sprawiający, że wyzwalacz będzie uruchamiany tylko wtedy, jeżeli wartość kolumny będzie mniejsza niż określona wartość.

Przygotowania do przykładu wyzwalacza

Jak zostało to wspomniane, wyzwalacze przydają się przy wykonywaniu zaawansowanych inspekcji zmian wprowadzonych do wartości kolumn. W kolejnym podrozdziale zostanie zaprezentowany wyzwalacz rejestrujący obniżenie ceny produktu o więcej niż 25 procent. Jeżeli wystąpi taki przypadek, wyzwalacz doda wiersz do tabeli `product_price_audit`. Jest ona tworzona przez skrypt `store_schema.sql` za pomocą następującej instrukcji:

```
CREATE TABLE product_price_audit (
    product_id INTEGER
        CONSTRAINT price_audit_fk_products
        REFERENCES products(product_id),
    old_price NUMBER(5, 2),
    new_price NUMBER(5, 2)
);
```

Jak widzimy, kolumna `product_id` tabeli `product_price_audit` jest kluczem obycym kolumny `product_id` tabeli `products`. W kolumnie `old_price` będzie składowana cena produktu przed wprowadzeniem zmiany, a w kolumnie `new_price` — cena po wprowadzeniu zmiany.

Tworzenie wyzwalacza

Do tworzenia wyzwalaczy służy instrukcja `CREATE TRIGGER`. Jej uproszczona składnia to:

```
CREATE [OR REPLACE] TRIGGER nazwa_wyzwalacza
[BEFORE | AFTER | INSTEAD OF | FOR] zdarzenie_wyzwalacza
ON nazwa_tabeli
[FOR EACH ROW]
[FORWARD | REVERSE] CROSSEDITION
[FORWARDS | PRECEDES] schemat.inny_wyzwalacz
[ENABLE | DISABLE]
[WHEN warunek_wyzwalacza]
BEGIN
    treść_wyzwalacza
END nazwa_wyzwalacza;
```

gdzie:

- `OR REPLACE` oznacza, że wyzwalacz ma zastąpić istniejący.
- `nazwa_wyzwalacza` jest nazwą wyzwalacza.
- `BEFORE` oznacza, że wyzwalacz jest uruchamiany przed wykonaniem zdarzenia wyzwalacza. `AFTER` oznacza, że wyzwalacz jest wykonywany po wykonaniu zdarzenia wyzwalacza. `INSTEAD OF` oznacza, że wyzwalacz jest wykonywany zamiast wykonania zdarzenia wyzwalacza. `FOR` (wprowadzone w Oracle Database 11g) umożliwia utworzenie złożonego wyzwalacza zawierającego cztery sekcje w treści.
- `zdarzenie_wyzwalacza` jest zdarzeniem uruchamiającym wyzwalacz.
- `nazwa_tabeli` jest nazwą tabeli, której dotyczy wyzwalacz.

- FOR EACH ROW oznacza, że jest tworzony wyzwalacz poziomu wierszy, czyli że po uruchomieniu wyzwalacza kod znajdujący się w *treść_wyzwalacza* będzie wykonywany raz dla każdego wiersza. Jeżeli pominiemy FOR EACH ROW, zostanie utworzony wyzwalacz poziomu instrukcji, co oznacza, że po uruchomieniu wyzwalacza kod znajdujący się w *treść_wyzwalacza* zostanie wykonany tylko raz.
- {FORWARD | REVERSE} CROSSDITION są opcjami wprowadzonymi w Oracle Database 11g z myślą o administratorach baz danych i aplikacji. Wyzwalacz międzyedycyjny FORWARD CROSSITION jest uruchamiany, gdy instrukcja DML dokonuje zmian w bazie danych, podczas gdy sieciowa aplikacja uzyskująca dostęp do bazy danych jest *poprawiana lub aktualizowana* (jest to ustawienie domyślne). Kod w treści wyzwalacza musi być zaprojektowany do obsługi zmian DML, gdy wprowadzanie poprawek lub aktualizacja aplikacji zostanie zakończona. Wyzwalacz międzyedycyjny REVERSE CROSSITION jest podobny, jest jednak przeznaczony do uruchamiania i obsługi zmian DML wprowadzonych po *poprawieniu lub aktualizowaniu aplikacji sieciowej*.
- {FOLLOWS | PRECEDES} *schemat.inny_wyzwalacz* jest opcją wprowadzoną w Oracle Database 11g i określa, czy uruchomienie wyzwalacza odbywa się przed uruchomieniem innego wyzwalacza określonego przez *schemat.inny_wyzwalacz*, czy też po nim. Można utworzyć serię wyzwalaczy uruchamianych w określonej kolejności.
- {ENABLE | DISABLE} jest opcją wprowadzoną w Oracle Database 11g i określa, czy po utworzeniu wyzwalacz jest początkowo włączony, czy też wyłączony (domyślnym ustawieniem jest ENABLE, czyli włączony). Wyłączony wyzwalacz można włączyć za pomocą instrukcji ALTER TRIGGER *nazwa_wyzwalacza* ENABLE lub włączając wszystkie wyzwalacze dla tabeli za pomocą instrukcji ALTER TABLE *nazwa_tabeli* ENABLE ALL TRIGGERS.
- *warunek_wyzwalacza* jest logicznym warunkiem określającym, czy kod wyzwalacza zostanie wykonany.
- *treść_wyzwalacza* zawiera kod wyzwalacza.

Przykładowy wyzwalacz prezentowany w tym podrozdziale będzie uruchamiany przed aktualizacją kolumny price w tabeli products. Będzie on ponadto używał wartości z tej kolumny zarówno przed wykonaniem instrukcji UPDATE, jak i po nim, będzie więc wyzwalaczem poziomu wierszy. Poza tym, ponieważ chcemy dokonać obserwacji, gdy cena zostanie obniżona o więcej niż 25 procent, musimy określić warunek wyzwalacza porównujący nową cenę ze starą. Poniższa instrukcja tworzy wyzwalacz before_product_price_update:

```
CREATE TRIGGER before_product_price_update
BEFORE UPDATE OF price
ON products
FOR EACH ROW WHEN (new.price < old.price * 0.75)
BEGIN
    dbms_output.put_line('product_id = ' || :old.product_id);
    dbms_output.put_line('Stara cena = ' || :old.price);
    dbms_output.put_line('Nowa cena = ' || :new.price);
    dbms_output.put_line('Obniżono cenę o ponad 25%');

    -- wstaw wiersz do tabeli product_price_audit
    INSERT INTO product_price_audit (
        product_id, old_price, new_price
    ) VALUES (
        :old.product_id, :old.price, :new.price
    );
END before_product_price_update;
/
```

W tej instrukcji należy zwrócić uwagę na pięć elementów:

- BEFORE UPDATE OF price oznacza, że wyzwalacz będzie uruchamiany przed modyfikacją kolumny price.
- FOR EACH ROW oznacza, że jest to wyzwalacz poziomu wierszy, czyli kod znajdujący się między słowami kluczowymi BEGIN i END będzie wykonywany dla każdego wiersza modyfikowanego przez instrukcję UPDATE.

- Warunek wyzwalacza jest określony jako (`new.price < old.price * .75`), co oznacza, że wyzwalacz będzie uruchamiany tylko wtedy, gdy nowa cena będzie mniejsza niż 75 procent starej ceny (czyli gdy cena zostanie obniżona o ponad 25 procent).
- W wyzwalaczu dostęp do nowej i starej wartości kolumny jest uzyskiwany za pomocą aliasów `:new` i `:old`.
- Kod wyzwalacza powoduje wyświetlenie `product_id`, nowej i starej wartości kolumny `price` oraz komunikatu o obniżce ceny o więcej niż 25 procent. Następnie do tabeli `product_price_audit` jest wstawiany wiersz zawierający `product_id` oraz starą i nową cenę.

Uruchamianie wyzwalacza

W celu wyświetlenia wyjścia z wyzwalacza należy uruchomić polecenie `SET SERVEROUTPUT ON`:

```
SET SERVEROUTPUT ON
```

Aby uruchomić wyzwalacz `before_product_price_update`, należy zmniejszyć cenę produktu o więcej niż 25 procent. W tym celu można na przykład uruchomić poniższą instrukcję obniżającą cenę produktów nr 5 i 10 o 30 procent (przez pomnożenie wartości kolumny `price` razy 0,7):

```
UPDATE products
SET price = price * .7
WHERE product_id IN (5, 10);
```

```
product_id = 5
Stara cena = 49,99
Nowa cena = 34,99
Obniżono cenę o ponad 25%
product_id = 10
Stara cena = 15,99
Nowa cena = 11,19
Obniżono cenę o ponad 25%
```

2 wierszy zostało zmodyfikowanych.

Jak widać, wyzwalacz został uruchomiony dla produktów nr 5 i 10.

Możemy się przekonać, że do tabeli `product_price_audit` zostały dodane dwa wiersze zawierające identyfikatory produktów oraz stare i nowe ceny:

```
SELECT *
FROM product_price_audit
ORDER BY product_id;
```

PRODUCT_ID	OLD_PRICE	NEW_PRICE
5	49,99	34,99
10	15,99	11,19

Uzyskiwanie informacji o wyzwalaczach

Informacje o utworzonych wyzwalaczach możemy uzyskać z perspektywy `user_triggers`. W tabeli 12.3 opisano jej wybrane kolumny.

Poniższa instrukcja pobiera z perspektywy `user_triggers` informacje o wyzwalaczu `before_product_price_update` z tabeli `user_triggers` (dla poprawy przejrzystości wyniki zostały sformatowane):

```
SET LONG 1000
SET PAGESIZE 100
SET LINESIZE 100

SELECT trigger_name, trigger_type, triggering_event, table_owner,
base_object_type, table_name, referencing_names, when_clause, status,
description, action_type, trigger_body
FROM user_triggers
```

Tabela 12.3. Wybrane kolumny perspektywy user_triggers

Kolumna	Typ	Opis
TRIGGER_NAME	VARCHAR2(128)	Nazwa wyzwalacza
TRIGGER_TYPE	VARCHAR2(16)	Typ wyzwalacza
TRIGGERING_EVENT	VARCHAR2(246)	Zdarzenie uruchamiające wyzwalacz
TABLE_OWNER	VARCHAR2(128)	Właściciel tabeli, do której odwołuje się wyzwalacz
BASE_OBJECT_TYPE	VARCHAR2(18)	Typ obiektu, do którego odwołuje się wyzwalacz
TABLE_NAME	VARCHAR2(128)	Nazwa tabeli, do której odwołuje się wyzwalacz
COLUMN_NAME	VARCHAR2(4000)	Nazwa kolumny, do której odwołuje się wyzwalacz
REFERENCING_NAMES	VARCHAR2(128)	Nazwa starych i nowych aliasów
WHEN_CLAUSE	VARCHAR2(4000)	Warunek wyzwalacza określający, kiedy jego kod zostanie wykonany
STATUS	VARCHAR2(8)	Okręsza, czy wyzwalacz jest włączony, czy wyłączony (ENABLED lub DISABLED)
DESCRIPTION	VARCHAR2(4000)	Opis wyzwalacza
ACTION_TYPE	VARCHAR2(11)	Typ akcji wyzwalacza (CALL lub PL/SQL)
TRIGGER_BODY	LONG	Kod znajdujący się w treści wyzwalacza. (Typ LONG umożliwia składowanie dużych ilości tekstu. Zostanie on opisany w rozdziale 14.)

```
WHERE trigger_name = 'BEFORE_PRODUCT_PRICE_UPDATE';
```

```
TRIGGER_NAME          TRIGGER_TYPE
```

```
-----
```

```
BEFORE_PRODUCT_PRICE_UPDATE BEFORE EACH ROW
```

```
TRIGGERING_EVENT
```

```
-----
```

```
UPDATE
```

```
TABLE_OWNER      BASE_OBJECT_TYPE   TABLE_NAME
```

```
-----
```

```
STORE           TABLE            PRODUCTS
```

```
REFERENCING_NAMES
```

```
-----
```

```
REFERENCING NEW AS NEW OLD AS OLD
```

```
WHEN_CLAUSE
```

```
-----
```

```
new.price < old.price * 0.75
```

```
STATUS
```

```
-----
```

```
ENABLED
```

```
DESCRIPTION
```

```
-----
```

```
before_product_price_update
```

```
BEFORE UPDATE OF
```

```
    price
```

```
ON
```

```
    products
```

```
FOR EACH ROW
```

```
ACTION_TYPE
```

```
-----
```

```
PL/SQL
```

```

TRIGGER_BODY
-----
BEGIN
  dbms_output.put_line('product_id = ' || :old.product_id);
  dbms_output.put_line('Stara cena = ' || :old.price);
  dbms_output.put_line('Nowa cena = ' || :new.price);
  dbms_output.put_line('Obniżono ceny o ponad 25%');

  -- Wstawienie wiersza do tabeli product_price_audit
  INSERT INTO product_price_audit (
    product_id, old_price, new_price
  ) VALUES (
    :old.product_id, :old.price, :new.price
  );
END before_product_price_update;

```



Informacje na temat wszystkich wyzwalaczy, do jakich masz dostęp, możesz uzyskać za pomocą instrukcji `all_triggers`.

Uwaga

Włączanie i wyłączanie wyzwalacza

Uruchamianie wyzwalacza można wyłączyć za pomocą instrukcji `ALTER TRIGGER`. Poniższa instrukcja wyłącza wyzwalacz `before_product_price_update`:

```
ALTER TRIGGER before_product_price_update DISABLE;
```

Kolejna instrukcja włącza wyzwalacz `before_product_price_update`:

```
ALTER TRIGGER before_product_price_update ENABLE;
```

Usuwanie wyzwalacza

Do usuwania wyzwalaczy służy instrukcja `DROP TRIGGER`. Poniższa instrukcja usuwa wyzwalacz `before_product_price_update`:

```
DROP TRIGGER before_product_price_update;
```

Rozszerzenia PL/SQL

W tym podrozdziale poznasz wybrane rozszerzenia PL/SQL. Będą to:

- typ `SIMPLE_INTEGER`, wprowadzony w Oracle Database 11g,
- obsługa sekwencji w PL/SQL, wprowadzona w Oracle Database 11g,
- generowanie natywnego kodu maszynowego z PL/SQL, wprowadzone w Oracle Database 11g,
- możliwość użycia w PL/SQL klauzuli `WITH`, wprowadzona w Oracle Database 12c.

Typ SIMPLE_INTEGER

Typ `SIMPLE_INTEGER` jest podtypem `BINARY_INTEGER`. Może składować taki sam zakres wartości jak `BINARY_INTEGER`, z tą różnicą, że typ `SIMPLE_INTEGER` nie może składować wartości `NULL`. Zakres wartości, jakie typ `SIMPLE_INTEGER` może przechować, zawiera się w przedziale od `-2 147 483 647` do `1 147 483 647`.

Gdy są stosowane wartości typu `SIMPLE_INTEGER`, przepelnienie arytmetyczne jest obcinane, dlatego też nie jest zgłoszany błąd, jeżeli wystąpi ono podczas obliczeń. Ponieważ błędy przepelniania są ignorowane, `SIMPLE_INTEGER` może zawijać się z liczbą dodatniej w ujemną i odwrotnie, na przykład:

$$2^{30} + 2^{30} = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}$$

$$-2^{31} + -2^{31} = 0x80000000 + 0x80000000 = 0x00000000 = 0$$

W pierwszym przykładzie są dodawane dwie liczby dodatnie, dając wynik ujemny. W drugim przykładzie dodawane są dwie liczby ujemne, dając w wyniku zero.

Ponieważ przepelnienie jest ignorowane i obcinane, gdy w obliczeniach są stosowane wartości typu SIMPLE_INTEGER, ten typ oferuje lepszą wydajność niż BINARY_INTEGER, jeśli administrator skonfiguruje bazę danych do komplikacji PL/SQL do natywnego kodu maszynowego. Jeżeli nie chcemy składować wartości NULL i przycinanie przepelnienia w obliczeniach nie ma znaczenia, należy stosować typ SIMPLE_INTEGER.

W poniższej procedurze `get_area()` zademonstrowano użycie typu SIMPLE_INTEGER. Ta procedura oblicza i wypisuje pole prostokąta:

```
CREATE PROCEDURE get_area
AS
  v_width SIMPLE_INTEGER := 10;
  v_height SIMPLE_INTEGER := 2;
  v_area SIMPLE_INTEGER := v_width * v_height;
BEGIN
  DBMS_OUTPUT.PUT_LINE('v_area = ' || v_area);
END get_area;
/
```

 Przykłady prezentowane w tym rozdziale są zamieszczone w skrypcie `plsql_11g_examples.sql` znajdującym się w katalogu `SQL`. Skrypt może zostać uruchomiony w Oracle Database 11g.

Poniższy przykład przedstawia uruchomienie procedury `get_area()`:

```
SET SERVEROUTPUT ON
CALL get_area();
v_area = 20
```

Zgodnie z oczekiwaniemi obliczone pole wynosi 20.

Sekwencje w PL/SQL

W poprzednim rozdziale pokazano, jak tworzyć i wykorzystywać sekwencje liczb w SQL. W Oracle Database 11g i nowszych sekwencje mogą być również używane w kodzie PL/SQL.

Jak pamiętamy, sekwencje generują serie liczb. Tworząc sekwencję w SQL, określamy jej początkową wartość oraz przyrost kolejnych liczb. Do uzyskania bieżącej wartości z sekwencji wykorzystujemy pseudokolumnę `currval`, a do uzyskania wartości kolejnej — pseudokolumnę `nextval`. Przed siegnięciem po wartość z `currval`, musimy użyć `nextval` w celu wygenerowania początkowej liczby.

Poniższa instrukcja tworzy tabelę o nazwie `new_products`, którą wkrótce wykorzystamy:

```
CREATE TABLE new_products (
  product_id INTEGER CONSTRAINT new_products_pk PRIMARY KEY,
  name VARCHAR2(30) NOT NULL,
  price NUMBER(5, 2)
);
```

Kolejna instrukcja tworzy sekwencję o nazwie `s_product_id`:

```
CREATE SEQUENCE s_product_id;
```

Poniższa instrukcja tworzy procedurę `add_new_products`, która wykorzystuje sekwencję `s_product_id` do zapisania wartości w kolumnie `product_id` wiersza dodanego do tabeli `new_products`. Należy zauważyć użycie pseudokolumn `nextval` i `currval` w kodzie PL/SQL (jest to nowość w Oracle Database 11g):

```
CREATE PROCEDURE add_new_products
AS
  v_product_id BINARY_INTEGER;
BEGIN
  -- używa nextval do wygenerowania początkowej liczby w sekwencji
  v_product_id := s_product_id.nextval;
  DBMS_OUTPUT.PUT_LINE('v_product_id = ' || v_product_id);

  -- wstawia wiersz do tabeli new_products
  INSERT INTO new_products
  VALUES (v_product_id, 'Fizyka plazmy', 49.95);
```

```

DBMS_OUTPUT.PUT_LINE('s_product_id.currvval = ' || s_product_id.currvval);

-- używa nextval do wygenerowania kolejnej liczby w sekwencji
v_product_id := s_product_id.nextval;
DBMS_OUTPUT.PUT_LINE('v_product_id = ' || v_product_id);

-- wstawia kolejny wiersz do tabeli new_products
INSERT INTO new_products
VALUES (v_product_id, 'Fizyka kwantowa', 69.95);

DBMS_OUTPUT.PUT_LINE('s_product_id.currvval = ' || s_product_id.currvval);
END add_new_products;
/

```

Poniżej zostało przedstawione uruchomienie procedury oraz zawartość tabeli `new_products`:

```

SET SERVEROUTPUT ON
CALL add_new_products();
v_product_id = 1
s_product_id.currvval = 1
v_product_id = 2
s_product_id.currvval = 2

SELECT * FROM new_products;

PRODUCT_ID NAME PRICE
----- -----
1 Fizyka plazmy 49,95
2 Fizyka kwantowa 69,95

```

Zgodnie z oczekiwaniemi do tabeli zostały dodane dwa wiersze.

Generowanie natywnego kodu maszynowego z PL/SQL

Domyślnie każda jednostka programu PL/SQL jest komplikowana do formy pośredniej kodu zrozumiałego dla komputera. Ten kod jest składowany w bazie danych i interpretowany przy każdorazowym uruchamianiu.

Podczas natywnej komplikacji PL/SQL kod PL/SQL jest przekształcany w kod natywny i zapisywany w bibliotekach współdzielonych. Kod natywny jest wykonywany znacznie szybciej niż kod pośredni, ponieważ nie musi zostać zinterpretowany przed wykonaniem.

W niektórych wersjach baz danych wcześniejszych niż Oracle Database 11g istnieje możliwość komplikacji kodu PL/SQL do kodu w języku C, który można później skompilować do kodu maszynowego. Jest to proces żmudny i problematyczny. W Oracle Database 11g i nowszych możemy bezpośrednio wygenerować kod maszynowy.

Konfigurowanie bazy danych do generowania natywnego kodu maszynowego powinno być wykonywane jedynie przez doświadczonych administratorów (w związku z czym wykracza poza zakres niniejszej książki). Wszystkie informacje dotyczące generowania natywnego kodu maszynowego z PL/SQL można znaleźć w podręczniku *PL/SQL User's Guide and Reference* wydanym przez Oracle Corporation.

Klauzula WITH

Nowością w Oracle Database 12c jest możliwość użycia klauzuli `WITH` w PL/SQL. Klauzula `WITH` umożliwia definiowanie wbudowanych funkcji PL/SQL, do których następnie można odwołać się w zapytaniu.

Poniższe zapytanie oblicza pole koła:

```

SET SERVEROUTPUT ON
WITH
FUNCTION circle_area(p_radius NUMBER) RETURN NUMBER IS
  v_pi NUMBER := 3.1415926;
  v_area NUMBER;

```

```

BEGIN
  v_area := v_pi * POWER(p_radius, 2);
  RETURN v_area;
END;
SELECT circle_area(3) FROM dual
/

```

CIRCLE_AREA(3)

28,274334

Kolejny przykład oblicza średnią cenę produktu:

```

WITH
FUNCTION average_product_price (
  p_product_type_id IN INTEGER
) RETURN NUMBER AS
  v_average_product_price NUMBER;
BEGIN
  SELECT AVG(price)
  INTO v_average_product_price
  FROM products
  WHERE product_type_id = p_product_type_id;
  RETURN v_average_product_price;
END;
SELECT average_product_price(1) FROM dual
/

```

AVERAGE_PRODUCT_PRICE(1)

24,965

Instrukcja `SELECT` z przykładu przekazuje parametr o wartości 1 jako typ produktu do `average_product_price()`. Taka wartość parametru opisuje książki. Przykład zwraca średnią cenę produktu typu książki.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- programy PL/SQL są podzielone na bloki zawierające instrukcje PL/SQL i SQL,
- pętla, taka jak `WHILE` lub `FOR`, wykonuje instrukcje kilka razy,
- w PL/SQL kursor umożliwia odczyt wierszy zwróconych przez zapytanie,
- wyjątki są używane do obsługi błędów powstających podczas wykonywania kodu PL/SQL,
- procedura zawiera grupę instrukcji. Procedury umożliwiają skoncentrowanie logiki biznesowej w bazie danych i mogą być uruchamiane przez dowolny program uzyskujący dostęp do bazy danych,
- funkcja jest podobna do procedury, musi jednak zwracać wartość,
- procedury i funkcje mogą być grupowane w pakiety hermetyzujące powiązane z sobą funkcjonalności w odrębnej jednostce,
- wyzwalacz jest procedurą wykonywaną automatycznie przez bazę danych, gdy zostanie uruchomiona określona instrukcja `INSERT`, `UPDATE` lub `DELETE`.

W następnym rozdziale zajmiemy się obiektami bazy danych.

ROZDZIAŁ

13

Obiekty bazy danych

W tym rozdziale:

- dowiesz się, jak tworzyć typy obiektowe zawierające atrybuty i metody,
- użyjemy typów obiektowych do definiowania obiektów kolumnowych i tabel obiektowych,
- dowiesz się, jak modyfikować obiekty w SQL i PL/SQL,
- nauczysz się tworzyć hierarchie typów,
- zdefiniujesz własne konstruktory ustawiające atrybuty obiektu,
- dowiesz się, jak przesłonić metodę z jednego typu metodą z innego typu.

Wprowadzenie do obiektów

Języki programowania zorientowane obiektowo, takie jak Java, C++ i C#, umożliwiają definiowanie klas stanowiących szablony obiektów. Klassy definiują atrybuty i metody. Atrybuty są wykorzystywane do przechowywania stanu obiektu, a metody służą do modelowania jego zachowań.

Wraz z wydaniem Oracle Database 8 obiekty stały się dostępne w bazie danych, a w kolejnych wydaniach właściwości obiektowe zostały znacznie ulepszone. Udostępnienie obiektów w bazie danych było przełomem, ponieważ umożliwiają one definiowanie własnych klas zwanych **typami obiektowymi**. Typy obiektowe w bazie danych, podobnie jak klasy w Javie i C#, zawierają atrybuty i metody. Typy obiektowe są czasem nazywane typami definiowanymi przez użytkownika.

Prostym przykładem typu obiektowego jest typ reprezentujący produkt. Taki typ obiektowy mógłby zawierać atrybuty dla nazwy produktu, jego opisu, ceny i w przypadku produktów psujących się — okresu, przez jaki mogą być przechowywane na półce sklepowej. Móglby również zawierać metodę zwracającą ostateczny termin sprzedaży, obliczany na podstawie możliwego okresu przechowywania i bieżącej daty.

Innym przykładem typu obiektowego jest typ reprezentujący osobę. Taki typ obiektowy może zawierać atrybuty dla imienia i nazwiska, daty urodzenia i adresu zamieszkania. Adres zamieszkania mógłby być również reprezentowany przez typ obiektowy składający takie informacje, jak ulica, miasto, województwo i kod pocztowy.

W tym rozdziale będziemy pracowali z typami obiektowymi reprezentującymi produkt, osobę i adres. Dowiesz się również, jak z tych typów obiektowych tworzyć tabele, umieszczać w tych tabelach faktyczne obiekty oraz jak nimi manipulować z użyciem języków SQL i PL/SQL.

Uruchomienie skryptu tworzącego schemat bazy danych object_schema

W katalogu SQL znajduje się skrypt *object_schema.sql*, tworzący konto użytkownika *object_user* z hasłem *object_password*. Ten skrypt tworzy również typy i tabele, wykonuje instrukcje INSERT, a także generuje kod PL/SQL prezentowany w pierwszej części rozdziału.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika **system**.
3. Uruchomić skrypt *object_schema.sql* w SQL*Plus za pomocą polecenia **@**.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu **C:\SQL**, to należy wpisać polecenie:

```
@ C:\SQL\object_schema.sql
```

Po zakończeniu pracy skryptu zostanie zalogowany użytkownik *object_user*.

Tworzenie typów obiektowych

Do tworzenia typów obiektowych służy instrukcja **CREATE TYPE**. W poniższym przykładzie za pomocą tej instrukcji tworzony jest typ obiektowy o nazwie *t_address*. Reprezentuje on adres i zawiera atrybuty o nazwach: *street*, *city*, *state* i *zip*:

```
CREATE TYPE t_address AS OBJECT (
    street VARCHAR2(15),
    city VARCHAR2(15),
    state CHAR(3),
    zip VARCHAR2(5)
);
/
```

Każdy atrybut jest definiowany z użyciem typu bazodanowego. Na przykład atrybut *street* jest definiowany jako *VARCHAR2(15)*. Jak się wkrótce przekonamy, typem atrybutu może być również typ obiektowy.

W kolejnym przykładzie jest tworzony obiekt o nazwie *t_person*. Ma on atrybut *address*, którego typem jest *t_address*:

```
CREATE TYPE t_person AS OBJECT (
    id INTEGER,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    dob DATE,
    phone VARCHAR2(12),
    address t_address
);
/
```

W kolejnym przykładzie jest tworzony obiekt o nazwie *t_product*, który będzie używany do reprezentowania produktów. Należy zauważyć, że za pomocą klauzuli **FUNCTION** w tym typie deklarowana jest funkcja o nazwie *get_sell_by_date()*:

```
CREATE TYPE t_product AS OBJECT (
    id INTEGER,
    name VARCHAR2(10),
    description VARCHAR2(22),
    price NUMBER(5, 2),
    days_valid INTEGER,
```

```
-- get_sell_by_date() zwraca datę, przed którą
-- produkt musi zostać sprzedany
MEMBER FUNCTION get_sell_by_date RETURN DATE
);
/

```

Ponieważ `t_product` zawiera deklarację metody, konieczne jest również utworzenie *treści* `t_product`. Zawiera ona kod metody i jest tworzona za pomocą instrukcji `CREATE TYPE BODY`. W poniższym przykładzie jest tworzona treść dla obiektu `t_product`. W treści znajduje się kod funkcji `get_sell_by_date`:

```
CREATE TYPE BODY t_product AS
-- get_sell_by_date() zwraca datę, przed którą
-- produkt musi zostać sprzedany
MEMBER FUNCTION get_sell_by_date RETURN DATE IS
    v_sell_by_date DATE;
BEGIN
    -- oblicza termin sprzedaży poprzez dodanie wartości atrybutu days_valid
    -- do bieżącej daty (SYSDATE)
    SELECT days_valid + SYSDATE
    INTO v_sell_by_date
    FROM dual;

    -- zwraca ostateczny termin sprzedaży
    RETURN v_sell_by_date;
END;
END;
/

```

Funkcja `get_sell_by_date` oblicza i zwraca datę, przed którą musi zostać sprzedany produkt. W tym celu do bieżącej daty zwracanej przez wbudowaną funkcję bazy danych `SYSDATE()` jest dodawana wartość atrybutu `days_valid`.

Dla typu możemy również utworzyć synonim publiczny, co umożliwia wszystkim użytkownikom definiowanie za jego pomocą kolumn we własnych tabelach. W poniższym przykładzie jest tworzony publiczny synonim `t_pub_product` dla obiektu `t_product`:

```
CREATE PUBLIC SYNONYM t_pub_product FOR t_product;
```

Uzyskiwanie informacji o typach obiektowych za pomocą DESCRIBE

Informacje o typie obiektowym można uzyskać za pomocą polecenia `DESCRIBE`. Poniższe przykłady zawierają opis typów `t_address`, `t_person` i `t_product`:

DESCRIBE t_address

Name	NULL?	Type
STREET		VARCHAR2(15)
CITY		VARCHAR2(15)
STATE		CHAR(3)
ZIP		VARCHAR2(5)

DESCRIBE t_person

Name	NULL?	Type
ID		NUMBER(38)
FIRST_NAME		VARCHAR2(10)
LAST_NAME		VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)
ADDRESS		T_ADDRESS

DESCRIBE t_product

Name	NULL?	Type
ID		NUMBER(38)
NAME		VARCHAR2(10)
DESCRIPTION		VARCHAR2(22)
PRICE		NUMBER(5,2)
DAYS_VALID		NUMBER(38)

METHOD

MEMBER FUNCTION GET_SELL_BY_DATE RETURNS DATE

Za pomocą polecenia `SET DESCRIBE DEPTH` możemy ustawić poziom, do jakiego będą wyświetlane osadzone typy. W poniższym przykładzie głębokość jest ustawiana na 2 i jest wyświetlany opis typu `t_person`. Należy zauważyć, że zostały wyświetlane atrybuty `address` będącego osadzonym obiektem typu `t_address`:

`SET DESCRIBE DEPTH 2``DESCRIBE t_person`

Name	NULL?	Type
ID		NUMBER(38)
FIRST_NAME		VARCHAR2(10)
LAST_NAME		VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)
ADDRESS		T_ADDRESS
STREET		VARCHAR2(15)
CITY		VARCHAR2(15)
STATE		CHAR(3)
ZIP		VARCHAR2(5)

Użycie typów obiektowych w tabelach bazy danych

Wiesz już, jak tworzyć typy obiektowe, teraz nauczysz się używać ich w tabelach bazy danych. Za pomocą typu obiektowego definiuje się kolumnę. Obiekty składowane w tej kolumnie nazywamy **obiektami kolumnowymi**. Typ obiektowy może również zostać użyty do zdefiniowania całego wiersza w tabeli, którą wówczas nazywamy **tabelą obiektową**. Każdy obiekt w tabeli obiektowej ma unikatowy identyfikator obiektu (OID), który opisuje lokalizację obiektu w bazie danych. Do uzyskania dostępu do jednego wiersza w tabeli obiektowej możemy użyć **odwołania obiektowego**, które przypomina wskaźnik w języku C++. W tym podrozdziale zobaczyś przykłady obiektów, tabel oraz odwołań obiektowych.

Obiekty kolumnowe

Poniższy przykład tworzy tabelę o nazwie `products`, zawierającą kolumnę `product` o typie `t_product`. Ta tabela zawiera również kolumnę `quantity_in_stock`, w której będzie składowana liczba produktów w magazynie:

```
CREATE TABLE products (
    product      t_product,
    quantity_in_stock INTEGER
);
```

Wstawiając wiersz do tej tabeli, musimy użyć **konstruktora** w celu dostarczenia wartości atrybutów nowego obiektu `t_product`. Jak pamiętasz, typ `t_product` był tworzony za pomocą następującej instrukcji:

```
CREATE TYPE t_product AS OBJECT (
    id INTEGER,
    name VARCHAR2(15),
    description VARCHAR2(22),
    price NUMBER(5, 2),
    days_valid INTEGER,
```

```
-- get_sell_by_date() zwraca datę, przed którą
-- produkt musi zostać sprzedany
MEMBER FUNCTION get_sell_by_date RETURN DATE
);
/
```

Konstruktor jest metodą wbudowaną do typu obiektowego i ma taką samą nazwę jak typ obiektowy. Konstruktor przyjmuje parametry wykorzystywane do ustawiania atrybutów nowego obiektu. Konstruktor typu `t_product` ma nazwę `t_product` i przyjmuje pięć parametrów — po jednym dla każdego atrybutu. Na przykład `t_product(1, makaron, '0,5 kg paczka makaronu', 3.95, 10)` tworzy nowy obiekt `t_product` i ustawia jego atrybuty w następujący sposób:

- `id` na 1,
- `name` na `makaron`,
- `description` na `'0,5 kg paczka makaronu'`,
- `price` na `3.95`
- `days_valid` na `10`.

Poniższe instrukcje `INSERT` wstawiają dwa wiersze do tabeli `products`. Należy zauważyć użycie konstruktora `t_product` do dostarczenia wartości do obiektów kolumnowych `product`:

```
INSERT INTO products (
    product,
    quantity_in_stock
) VALUES (
    t_product(1, 'makaron', '0,5 kg paczka makaronu', 3.95, 10),
    50
);

INSERT INTO products (
    product,
    quantity_in_stock
) VALUES (
    t_product(2, 'sardynki', '100 g puszka sardynek', 2.99, 5),
    25
);
```

Poniższe zapytanie pobiera te wiersze z tabeli `products`. Atrybuty obiektów kolumnowych `product` są wyświetlane w konstruktorze `t_product`:

```
SELECT *
FROM products;

PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
QUANTITY_IN_STOCK
-----
T_PRODUCT(1, 'makaron', '0,5 kg paczka makaronu', 3.95, 10)
      50

T_PRODUCT(2, 'sardynki', '100 g puszka sardynek', 2.99, 5)
      25
```

Możemy również pobrać jeden obiekt kolumnowy. W tym celu musimy zapewnić alias tabeli, za pośrednictwem którego wybierzemy obiekt. Poniższe zapytanie pobiera z tabeli `products` produkt nr 1. Należy zwrócić uwagę na użycie aliasu `p` dla tabeli `products`, za pośrednictwem którego w klauzuli `WHERE` jest określany `id` obiektu `product`:

```
SELECT p.product
FROM products p
WHERE p.product.id = 1;
```

```
PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PRODUCT(1, 'makaron', '0,5 kg paczka makaronu', 3,95, 10)
```

W instrukcji SELECT kolejnego zapytania są jawnie wymienione atrybuty id, name, price i days_valid obiektu product oraz kolumna quantity_in_stock:

```
SELECT p.product.id, p.product.name,
       p.product.price, p.product.days_valid, p.quantity_in_stock
  FROM products p
 WHERE p.product.id = 1;
```

PRODUCT.ID	PRODUCT.NAME	PRODUCT.PRICE	PRODUCT.DAYS_VALID	QUANTITY_IN_STOCK
1	makaron	3,95	10	50

Typ obiektowy t_product zawiera funkcję o nazwie get_sell_by_date(), obliczającą i zwracającą datę, przed którą produkt musi zostać sprzedany. Funkcja dodaje wartość atrybutu days_valid do bieżącej daty uzyskiwanej z bazy danych za pomocą funkcji SYSDATE(). Funkcję get_sell_by_date() można wywołać, korzystając z aliasu tabeli, co obrazuje poniższe zapytanie, w którym aliasem tabeli products jest p:

```
SELECT p.product.get_sell_by_date()
  FROM products p;
```

P.PRODUCT
12/11/06
12/11/01

Oczywiście, daty mogą być inne, ponieważ są obliczane z użyciem funkcji SYSDATE(), zwracającej bieżącą datę.

Poniższa instrukcja UPDATE modyfikuje opis produktu nr 1. Ponownie użyto aliasu p:

```
UPDATE products p
SET p.product.description = '0,7 kg paczka makaronu'
WHERE p.product.id = 1;
```

1 row updated.

Poniższa instrukcja DELETE usuwa produkt nr 2:

```
DELETE FROM products p
WHERE p.product.id = 2;
```

1 row deleted.

ROLLBACK;

 Po uruchomieniu powyższych instrukcji UPDATE i DELETE należy również uruchomić instrukcję ROLLBACK, aby wyniki otrzymywane w przykładach były zgodne z prezentowanymi w książce.

Tabele obiektowe

Z pomocą typu obiektowego można również zdefiniować całą tabelę, którą w takiej sytuacji nazywamy tabelą obiektową. W poniższym przykładzie jest tworzona tabela obiektowa o nazwie object_products, w której będą składowane obiekty typu t_product. Użyto słowa kluczowego OF do określenia tabeli jako obiektowej o typie t_product:

```
CREATE TABLE object_products OF t_product;
```

Do tabeli obiektowej można wstawić wiersz na dwa sposoby: użyć konstruktora w celu dostarczenia wartości atrybutów albo przesyłać wartości tak, jakbyśmy przesyłali wartości kolumn w tabeli relacyjnej. Poniższa instrukcja INSERT wstawia wiersz do tabeli object_products za pomocą konstruktora obiektu t_product:

```
INSERT INTO object_products VALUES (
    t_product(1, 'makaron', '0,5 kg paczka makaronu', 3.95, 10)
);
```

Kolejna instrukcja `INSERT` pomija konstruktor `t_product`. Wartości atrybutów dla `t_product` są przesyłane tak jak wartości kolumn w tabeli relacyjnej:

```
INSERT INTO object_products (
    id, name, description, price, days_valid
) VALUES (
    2, 'sardynki', '100 g puszka sardynek', 2.99, 5
);
```

Poniższe zapytanie pobiera te wiersze z tabeli `object_products`:

```
SELECT *
FROM object_products;
```

ID	NAME	DESCRIPTION	PRICE	DAYS_VALID
1	makaron	0,5 kg paczka makaronu	3,95	10
2	sardynki	100 g puszka sardynek	2,99	5

W zapytaniu można również określić pojedyncze atrybuty obiektu, na przykład:

```
SELECT id, name, price
FROM object_products op
WHERE id = 1;
```

ID	NAME	PRICE
1	makaron	3,95

lub

```
SELECT op.id, op.name, op.price
FROM object_products op
WHERE op.id = 1;
```

ID	NAME	PRICE
1	makaron	3,95

Do wybrania wiersza z tabeli obiektowej można użyć wbudowanej funkcji bazy danych Oracle — `VALUE()`. Traktuje ona wiersz jak faktyczny obiekt i zwraca atrybuty obiektu w jego konstruktorze. Funkcja `VALUE()` przyjmuje parametr zawierający alias tabeli, co obrazuje poniższe zapytanie:

```
SELECT VALUE(op)
FROM object_products op;
```

VALUE(OP)(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
T_PRODUCT(1, 'makaron', '0,5 kg paczka makaronu', 3,95, 10)
T_PRODUCT(2, 'sardynki', '100 g puszka sardynek', 2,99, 5)

Możemy również określić atrybut obiektu za pomocą funkcji `VALUE()`:

```
SELECT VALUE(op).id, VALUE(op).name, VALUE(op).price
FROM object_products op;
```

VALUE(OP).ID	VALUE(OP).NAME	VALUE(OP).PRICE
1	makaron	3,95
2	sardynki	2,99

Poniższa instrukcja `UPDATE` zmienia opis produktu nr 1:

```
UPDATE object_products
SET description = '0,7 kg paczka makaronu'
```

```
WHERE id = 1;
```

1 row updated.

Poniższa instrukcja DELETE usuwa produkt nr 2:

```
DELETE FROM object_products
WHERE id = 2;
```

1 row deleted.

```
ROLLBACK;
```

Przejdźmy do bardziej złożonej tabeli. Poniższa instrukcja CREATE TABLE tworzy tabelę obiektową o nazwie `object_customers`, w której są składowane obiekty typu `t_person`:

```
CREATE TABLE object_customers OF t_person;
```

Typ `t_person` zawiera osadzony obiekt `t_address`. Typ ten był tworzony za pomocą następującej instrukcji:

```
CREATE TYPE t_person AS OBJECT (
    id INTEGER,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    dob DATE,
    phone VARCHAR2(12),
    address t_address
);
/
```

Poniższe instrukcje INSERT wstawiają dwa wiersze do tabeli `object_customers`. Pierwsza instrukcja wykorzystuje konstruktory dla `t_person` i `t_address`, druga natomiast pomija konstruktor `t_person`:

```
INSERT INTO object_customers VALUES (
    t_person(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
        t_address('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
    )
);

INSERT INTO object_customers (
    id, first_name, last_name, dob, phone,
    address
) VALUES (
    2, 'Sylwia', 'Zielona', '68/02/05', '800-555-1212',
    t_address('Wolności 3', 'Śródmieście', 'MAZ', '12345')
);
```

Poniższe zapytanie pobiera te wiersze z tabeli `object_customers`. Atrybuty osadzonego obiektu kolumnowego `address` zostały wypisane w konstruktorze `t_address`:

```
SELECT *
FROM object_customers;
```

ID	FIRST_NAME	LAST_NAME	DOB	PHONE	ADDRESS(STREET, CITY, STATE, ZIP)
1	Jan	Brązowy	55/02/01	800-555-1211	T_ADDRESS('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
2	Sylwia	Zielona	68/02/05	800-555-1212	T_ADDRESS('Wolności 3', 'Śródmieście', 'MAZ', '12345')

Kolejne zapytanie pobiera z tabeli `object_customers` informacje o kliencie nr 1. Należy zauważać użycie aliasu tabeli `oc`, za pośrednictwem którego jest określany atrybut `id` w klauzuli `WHERE`:

```

SELECT *
FROM object_customers oc
WHERE oc.id = 1;

ID FIRST_NAME LAST_NAME DOB PHONE
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
1 Jan Brązowy 55/02/01 800-555-1211
T_ADDRESS('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')

```

W kolejnym zapytaniu informacje o kliencie są pobierane na podstawie wartości atrybutu state obiektu kolumnowego address:

```

SELECT *
FROM object_customers oc
WHERE oc.address.state = 'MAZ';

ID FIRST_NAME LAST_NAME DOB PHONE
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
2 Sylwia Zielona 68/02/05 800-555-1212
T_ADDRESS('Wolności 3', 'Śródmieście', 'MAZ', '12345')

```

W kolejnym zapytaniu w instrukcji SELECT są jawnie wymienione atrybuty id, first_name i last_name klienta nr 1, a także atrybuty osadzonego obiektu kolumnowego address:

```

SELECT oc.id, oc.first_name, oc.last_name,
       oc.address.street, oc.address.city, oc.address.state, oc.address.zip
FROM object_customers oc
WHERE oc.id = 1;

ID FIRST_NAME LAST_NAME ADDRESS.STREET ADDRESS.CITY ADD ADDRE
-----
1 Jan Brązowy Kościuszki 23 Siemiatycze MAL 12345

```

Identyfikatory obiektów i odwołania obiektowe

Każdy obiekt w bazie danych ma unikatowy **identyfikator obiektu** (OID), który możemy pobrać za pomocą funkcji REF(). Na przykład poniższe zapytanie pobiera OID klienta nr 1 w tabeli object_customers:

```

SELECT REF(oc)
FROM object_customers oc
WHERE oc.id = 1;

REF(OC)
-----
0000280209AB730342710449CB967C2AB2708558199C96B0783E754CD898D1AD5C5398DFDF010002F80000

```

Ten długi łańcuch cyfr i liter to właśnie OID, który identyfikuje położenie obiektu w bazie danych. OID można składować w odwołaniu obiektowym i później uzyskać dostęp do obiektu, na który wskazuje to odwołanie. Odwołanie obiektowe, które przypomina wskaźnik z języka C++, wskazuje za pomocą OID na obiekt składowany w tabeli obiektowej. Odwołania obiektowe mogą być użyte do modelowania relacji między obiektami, a jak się przekonasz, w PL/SQL odwołania obiektowe mogą być użyte do uzyskiwania dostępu do obiektów.

Odwołanie obiektowe definiuje się za pomocą typu REF. Poniższa instrukcja tworzy tabelę o nazwie purchases, zawierającą dwie kolumny odwołujące się do obiektów: customer_ref i product_ref.

```

CREATE TABLE purchases (
    id      INTEGER PRIMARY KEY,
    customer_ref REF t_person SCOPE IS object_customers,
    product_ref REF t_product SCOPE IS object_products
);

```

Klauzula SCOPE IS ogranicza odwołanie obiektowe, aby wskazywało na obiekty w określonej tabeli. Na przykład kolumna `customer_ref` może wskazywać jedynie na obiekty w tabeli `object_customers`; kolumna `product_ref` może wskazywać jedynie na obiekty w tabeli `object_products`.

Jak zostało to wspomniane, każdy obiekt w tabeli obiektowej posiada unikatowy identyfikator obiektu (OID), który można pobrać za pomocą funkcji `REF()`. Na przykład poniższa instrukcja `INSERT` wstawia wiersz do tabeli `purchases`. W zapytaniach została użyta funkcja `REF()` w celu uzyskania identyfikatorów obiektów reprezentujących klienta nr 1 i produkt nr 1 w tabelach `object_customers` i `object_products`:

```
INSERT INTO purchases (
    id,
    customer_ref,
    product_ref
) VALUES (
    1,
    (SELECT REF(oc) FROM object_customers oc WHERE oc.id = 1),
    (SELECT REF(op) FROM object_products op WHERE op.id = 1)
);
```

W tym przykładzie rejestrujemy, że klient nr 1 kupił produkt nr 1.

Poniższe zapytanie wybiera wiersz z tabeli `purchases`. Należy zauważać, że kolumny `customer_ref` i `product_ref` zawierają odwołania do obiektów w tabelach `object_customers` i `object_products`:

```
SELECT *
FROM purchases;
```

ID

CUSTOMER_REF

PRODUCT_REF

1
0000220208AB730342710449CB967C2AB2708558199C96B0783E754CD898D1AD5C5398DFDF
0000220208429D7E94A9B646C190D7FEDE499BE34E309E83F994DF431892E0CD33E9391E15

Faktyczne obiekty składowane w odwołaniu obiektowym można pobrać za pomocą funkcji `DREF()`, która przyjmuje jako parametr odwołanie obiektu i zwraca faktyczny obiekt. Na przykład w poniższym zapytaniu użyto funkcji `DREF()` do pobrania informacji o kliencie nr 1 i produkcie nr 1 za pośrednictwem kolumn `customer_ref` i `product_ref` tabeli `purchases`:

```
SELECT DREF(customer_ref), DREF(product_ref)
FROM purchases;
```

DREF(CUSTOMER_REF)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY,

DREF(PRODUCT_REF)(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)

T_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211', T_ADDRESS('Kościuszki 23', 'Siemiatycze', 'MAL', '12345'))
T_PRODUCT(1, 'makaron', '0,5 kg paczka makarona', 3,95, 10)

Kolejne zapytanie pobiera atrybuty `first_name` i `address.street` klienta oraz atrybut `name` produktu:

```
SELECT DREF(customer_ref).first_name,
       DREF(customer_ref).address.street, DREF(product_ref).name
  FROM purchases;
```

DREF(CUST DREF(CUSTOMER_ DREF(PRODUCT_R

Jan Kościuszki 23 makaron

Poniższa instrukcja `UPDATE` zmienia kolumnę `product_ref`, aby wskazywała na produkt nr 2:

```
UPDATE purchases SET product_ref =
    (SELECT REF(op) FROM object_products op WHERE op.id = 2
```

```
) WHERE id = 1;
```

1 row updated.

Poniższe zapytanie weryfikuje wprowadzoną zmianę:

```
SELECT DEREF(customer_ref), DEREF(product_ref)
FROM purchases;
```

```
DREF(CUSTOMER_REF)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY,
-----
DREF(PRODUCT_REF)(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211', T_ADDRESS('Kościuszki 23', 'Siemiatycze',
'MAL', '12345'))
T_PRODUCT(2, 'sardynki', '100 g puszka sardynek', 2,99, 5)
```

Porównywanie wartości obiektów

Wartości dwóch obiektów w klauzuli WHERE zapytania możemy porównać za pomocą operatora równości (=). Na przykład poniższe zapytanie pobiera z tabeli object_customers informacje o kliencie nr 1:

```
SELECT oc.id, oc.first_name, oc.last_name, oc.dob
FROM object_customers oc
WHERE VALUE(oc) =
  t_person(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
  t_address('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
1	Jan	Brązowy	55/02/01

Kolejne zapytanie pobiera informacje o produkcie nr 1 z tabeli object_products:

```
SELECT op.id, op.name, op.price, op.days_valid
FROM object_products op
WHERE VALUE(op) = t_product(1, 'makaron', '0,5 kg paczka makaronu', 3.95, 10);
```

ID	NAME	PRICE	DAYS_VALID
1	makaron	3,95	10

W klauzuli WHERE można również użyć operatorów \leftrightarrow i IN:

```
SELECT oc.id, oc.first_name, oc.last_name, oc.dob
FROM object_customers oc
WHERE VALUE(oc) <>
  t_person(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
  t_address('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
2	Sylwia	Zielona	68/02/05

```
SELECT op.id, op.name, op.price, op.days_valid
FROM object_products op
WHERE VALUE(op) IN t_product(1, 'makaron', '0,5 kg paczka makaronu', 3.95, 10);
```

ID	NAME	PRICE	DAYS_VALID
1	makaron	3,95	10

Jeżeli chcemy użyć operatora takiego jak $<$, $>$, \leq , \geq , LIKE lub BETWEEN, musimy utworzyć funkcję odwzorowującą dla typu. Musi ona zwracać jedną wartość o wbudowanym typie, która może zostać użyta

360 Oracle Database 12c i SQL. Programowanie

do porównania dwóch obiektów. Wartość zwracana przez funkcję odwzorowującą będzie inna dla każdego typu obiektowego i musimy określić, który z atrybutów lub ich konkatenacji najlepiej reprezentuje wartość obiektu. Na przykład w przypadku typu `t_product` warto zwrócić atrybut `price`, a w przypadku typu `t_person` — konkatenację atrybutów `last_name` i `first_name`.

Poniższe instrukcje tworzą typ o nazwie `t_person2`, który zawiera funkcję odwzorowującą o nazwie `get_string()`. Zwraca ona napis `VARCHAR2`, zawierający konkatenację atrybutów `last_name` i `first_name`:

```
CREATE TYPE t_person2 AS OBJECT (
    id INTEGER,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    dob DATE,
    phone VARCHAR2(12),
    address t_address,
);

-- deklaracja funkcji odwzorowującej get_string()
-- zwracającej napis typu VARCHAR2
MAP MEMBER FUNCTION get_string RETURN VARCHAR2
);
/
CREATE TYPE BODY t_person2 AS
    -- definicja funkcji odwzorowującej get_string()
    MAP MEMBER FUNCTION get_string RETURN VARCHAR2 IS
        BEGIN
            -- zwraca napis będący konkatenacją
            -- atrybutów last_name i first_name
            RETURN last_name || ' ' || first_name;
        END get_string;
    END;
/

```

Jak się wkrótce przekonasz, podczas porównywania obiektów funkcja `get_string()` jest wywoływana automatycznie.

Poniższe instrukcje tworzą tabelę o nazwie `object_customers2` i wstawiają do niej wiersze:

```
CREATE TABLE object_customers2 OF t_person2;

INSERT INTO object_customers2 VALUES (
    t_person2(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
    t_address('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
)
;

INSERT INTO object_customers2 VALUES (
    t_person2(2, 'Sylwia', 'Zielona', '68/02/05', '800-555-1212',
    t_address('Wolności 3', 'Śródmieście', 'MAZ', '12345')
)
;
```

W klauzuli WHERE poniższego zapytania użyto operatora `>:`

```
SELECT oc2.id, oc2.first_name, oc2.last_name, oc2.dob
FROM object_customers2 oc2
WHERE VALUE(oc2) >
    t_person2(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
    t_address('Kościuszki 23', 'Siemiatycze', 'MAL', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
2	Sylwia	Zielona	68/02/05

Po uruchomieniu tego zapytania baza danych automatycznie wywołuje funkcję `get_string` w celu porównania obiektów z tabeli `object_customers2` z obiektem wymienionym za operatorem `>` w klauzuli `WHERE`. Funkcja `get_string()` zwraca konkatenację atrybutów `last_name` i `first_name` obiektów, a ponieważ po zinterpretowaniu napis Zielona Sylwia jest dłuższy od Brązowy Jan, jest on zwracany przez zapytanie.

Użycie obiektów w PL/SQL

W PL/SQL możemy tworzyć obiekty i manipulować nimi. W tym podrozdziale opisano użycie pakietu `product_package`, tworzonego przez skrypt `object_schema.sql`. Zawiera on następujące metody:

- funkcję `get_products()`, która zwraca REF CURSOR wskazujący na obiekty w tabeli `object_products`,
- procedurę `display_product()`, wyświetlającą atrybuty jednego obiektu z tabeli `object_products`,
- procedurę `insert_product()`, wstawiającą obiekt do tabeli `object_products`,
- procedurę `update_product_price()`, modyfikującą atrybut `price` obiektu z tabeli `object_products`,
- funkcję `get_product()`, zwracającą jeden obiekt z tabeli `object_products`,
- procedurę `update_product`, aktualizującą wszystkie atrybuty obiektu z tabeli `object_products`,
- funkcję `get_product_ref()`, zwracającą odwołanie do jednego obiektu z tabeli `object_products`,
- procedurę `delete_product()`, usuwającą jeden obiekt z tabeli `object_products`.

Skrypt `object_schema.sql` zawiera następującą specyfikację pakietu:

```
CREATE PACKAGE product_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_products RETURN t_ref_cursor;
  PROCEDURE display_product(
    p_id IN object_products.id%TYPE
  );
  PROCEDURE insert_product(
    p_id IN object_products.id%TYPE,
    p_name IN object_products.name%TYPE,
    p_description IN object_products.description%TYPE,
    p_price IN object_products.price%TYPE,
    p_days_valid IN object_products.days_valid%TYPE
  );
  PROCEDURE update_product_price(
    p_id IN object_products.id%TYPE,
    p_factor IN NUMBER
  );
  FUNCTION get_product(
    p_id IN object_products.id%TYPE
  ) RETURN t_product;
  PROCEDURE update_product(
    p_product t_product
  );
  FUNCTION get_product_ref(
    p_id IN object_products.id%TYPE
  ) RETURN REF t_product;
  PROCEDURE delete_product(
    p_id IN object_products.id%TYPE
  );
END product_package;
/
```

W kolejnych podrozdziałach zostaną opisane metody z treści pakietu `product_package`.

Funkcja `get_products()`

Funkcja `get_products()` zwraca REF CURSOR, wskazujący na obiekty w tabeli `object_products`. W treści pakietu `product_package` funkcja `get_products` jest definiowana w następujący sposób:

```

FUNCTION get_products
RETURN t_ref_cursor IS
-- deklaracja obiektu t_ref_cursor
  v_products_ref_cursor t_ref_cursor;
BEGIN
-- pobranie REF CURSOR
OPEN v_products_ref_cursor FOR
  SELECT VALUE(op)
    FROM object_products op
   ORDER BY op.id;

-- zwraca REF CURSOR
RETURN v_products_ref_cursor;
END get_products;

```

Poniższe zapytanie wywołuje funkcję `product_package.get_products()` w celu pobrania produktów z tabeli `object_products`:

```

SELECT product_package.get_products
FROM dual;

```

```

GET_PRODUCTS
-----
CURSOR STATEMENT : 1
CURSOR STATEMENT : 1
VALUE(OP)(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PRODUCT(1, 'makaron', '0,5 kg paczka makaronu', 3,95, 10)
T_PRODUCT(2, 'sardynki', '100 g puszka sardynek', 2,99, 5)

```

Procedura `display_product()`

Procedura `display_product` wyświetla atrybuty jednego obiektu z tabeli `object_products`. W treści pakietu `product_package` ta procedura jest definiowana w następujący sposób:

```

PROCEDURE display_product(
  p_id IN object_products.id%TYPE
) AS
-- deklaracja obiektu t_product o nazwie v_product
  v_product t_product;
BEGIN
-- próba uzyskania produktu i zapisania go w v_product
  SELECT VALUE(op)
    INTO v_product
   FROM object_products op
  WHERE id = p_id;

-- wyświetla atrybuty v_product
  DBMS_OUTPUT.PUT_LINE('v_product.id=' || 
    v_product.id);
  DBMS_OUTPUT.PUT_LINE('v_product.name=' || 
    v_product.name);
  DBMS_OUTPUT.PUT_LINE('v_product.description=' || 
    v_product.description);
  DBMS_OUTPUT.PUT_LINE('v_product.price=' || 
    v_product.price);
  DBMS_OUTPUT.PUT_LINE('v_product.days_valid=' || 
    v_product.days_valid);

-- wywołyje v_product.get_sell_by_date() i wyświetla datę
  DBMS_OUTPUT.PUT_LINE('sell_by_date=' || 
    v_product.get_sell_by_date());
END display_product;

```

W poniższym przykładzie jest wywoływana procedura `product_package.display_product(1)` w celu pobrania produktu nr 1 z tabeli `object_products`:

```
SET SERVEROUTPUT ON
CALL product_package.display_product(1);
v_product.id=1
v_product.name=makaron
v_product.description=0,5 kg paczka makaronu
v_product.price=3,95
v_product.days_valid=10
sell_by_date=12/11/07
```

Procedura insert_product()

Procedura `insert_product()` wstawia obiekt do tabeli `object_products`. Jest ona definiowana w następujący sposób w treści pakietu `product_package`:

```
PROCEDURE insert_product(
    p_id IN object_products.id%TYPE,
    p_name IN object_products.name%TYPE,
    p_description IN object_products.description%TYPE,
    p_price IN object_products.price%TYPE,
    p_days_valid IN object_products.days_valid%TYPE
) AS
    -- tworzy obiekt t_product o nazwie v_product
    v_product t_product :=
        t_product(
            p_id, p_name, p_description, p_price, p_days_valid
        );
BEGIN
    -- wstawia v_product do tabeli object_products
    INSERT INTO object_products VALUES (v_product);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END insert_product;
```

Poniższy przykład wywołuje procedurę `product_package.insert_product()` w celu wstawienia nowego obiektu do tabeli `object_products`:

```
CALL product_package.insert_product(3, 'salsa',
    'słoik sosu salsa 250 g', 1.50, 20);
```

Procedura update_product_price()

Procedura `update_product_price` modyfikuje atrybut `price` obiektu w tabeli `object_products`. W treści pakietu `product_package` ta procedura jest definiowana w następujący sposób:

```
PROCEDURE update_product_price(
    p_id IN object_products.id%TYPE,
    p_factor IN NUMBER
) AS
    -- deklaracja obiektu t_product o nazwie v_product
    v_product t_product;
BEGIN
    -- próba wybrania produktu do modyfikacji
    -- i zapisania go w v_product
    SELECT VALUE(op)
    INTO v_product
    FROM object_products op
    WHERE id = p_id
    FOR UPDATE;
```

```
-- wyświetla aktualną cenę v_product
DBMS_OUTPUT.PUT_LINE('v_product.price=' || v_product.price);

-- mnoży v_product.price przez p_factor
v_product.price := v_product.price * p_factor;
DBMS_OUTPUT.PUT_LINE('Nowa wartość v_product.price=' || v_product.price);

-- aktualizuje produkt w tabeli object_products
UPDATE object_products op
SET op = v_product
WHERE id = p_id;
COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END update_product_price;
```

W poniższym przykładzie za pomocą procedury `product_package.update_product_price()` jest zmieniana cena produktu nr 3 w tabeli `object_products`:

```
CALL product_package.update_product_price(3, 2.4)
v_product.price=1,5
Nowa wartość v_product.price=3,6
```

Funkcja get_product()

Funkcja `get_product()` zwraca jeden obiekt z tabeli `object_products`. W treści pakietu `product_package` jest ona definiowana w następujący sposób:

```
FUNCTION get_product(
  p_id IN object_products.id%TYPE
)
RETURN t_product IS
  -- deklaracja obiektu t_product o nazwie v_product
  v_product t_product;
BEGIN
  -- pobiera produkt i zapisuje go w v_product
  SELECT VALUE(op)
  INTO v_product
  FROM object_products op
  WHERE op.id = p_id;

  -- zwraca v_product
  RETURN v_product;
END get_product;
```

W poniższym zapytaniu użyto funkcji `product_package.get_product()` do pobrania produktu nr 3 z tabeli `object_products`:

```
SELECT product_package.get_product(3)
FROM dual;
-----  
PRODUCT_PACKAGE.GET_PRODUCT(3)(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)  
-----  
T_PRODUCT(3, 'salsa', 'słoik sosu salsa 250 g', 3,6, 20)
```

Procedura update_product()

Procedura `update_product()` modyfikuje wszystkie atrybuty obiektu z tabeli `object_products`. W treści pakietu `product_package` procedura ta jest definiowana w następujący sposób:

```
PROCEDURE update_product(
  p_product IN t_product
```

```

) AS
BEGIN
    -- aktualizuje produkt w tabeli object_products
    UPDATE object_products op
    SET op = p_product
    WHERE id = p_product.id;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product;

```

W poniższym przykładzie wywołano procedurę `update_product()`, aby zmodyfikować informacje o produkcie nr 3 w tabeli `object_products`:

```
CALL product_package.update_product(t_product(3, 'salsa',
    'słoik sosu salsa 500 g', 2.70, 15));
```

Funkcja `get_product_ref()`

Funkcja `get_product_ref()` zwraca odwołanie do jednego obiektu z tabeli `object_products`. Ta funkcja jest definiowana w treści pakietu `product_package` w następujący sposób:

```

FUNCTION get_product_ref(
    p_id IN object_products.id%TYPE
)
RETURN REF t_product IS
    -- deklaracja odwołania do t_product
    v_product_ref REF t_product;
BEGIN
    -- pobiera REF do produktu
    -- i zapisuje go w v_product_ref
    SELECT REF(op)
    INTO v_product_ref
    FROM object_products op
    WHERE op.id = p_id;

    -- zwraca v_product_ref
    RETURN v_product_ref;
END get_product_ref;

```

Poniższe zapytanie wywołuje `product_package.get_product_ref()` w celu pobrania odwołania do produktu nr 3 z tabeli `object_products`:

```
SELECT product_package.get_product_ref(3)
FROM dual;
```

```
PRODUCT_PACKAGE.GET_PRODUCT_REF(3)
```

```
-----
```

```
0000280209DEF099B694E041D2828BDA40DEE568E52C2FCADCC208424C98FABEFE96EDD2DF010002E50002
```

W kolejnym przykładzie ponownie wywołano `product_package.get_product_ref()`, tym razem jednak użyto funkcji `DEREF()` w celu pobrania faktycznego produktu:

```
SELECT DEREF(product_package.get_product_ref(3))
FROM dual;
```

```
DEREF(PRODUCT_PACKAGE.GET_PRODUCT_REF(3))(ID, NAME, DESCRIPTION, PRICE,
```

```
-----
```

```
T_PRODUCT(3, 'salsa', 'słoik sosu salsa 500 g', 2,7, 15)
```

Procedura `delete_product()`

Procedura `delete_product()` usuwa jeden obiekt z tabeli `object_product()`. W treści pakietu `product_package()` ta procedura jest definiowana w następujący sposób:

```

PROCEDURE delete_product(
    p_id IN object_products.id%TYPE
) AS
BEGIN
    -- usuń produkt
    DELETE FROM object_products op
    WHERE op.id = p_id;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END delete_product;

```

Poniżej jest wywoływana procedura `product_package.delete_product()` w celu usunięcia produktu nr 3 z tabeli `object_products`:

```
CALL product_package.delete_product(3)
```

Omówiliśmy już wszystkie metody z pakietu `product_package`. Zajmiemy się teraz dwiema procedurami — `product_lifecycle()` i `product_lifecycle2()` — wywołującymi różne metody z tego pakietu.

Procedura `product_lifecycle()`

Procedura `product_lifecycle` jest definiowana w następujący sposób:

```

CREATE PROCEDURE product_lifecycle AS
    -- deklaracja obiektu
    v_product t_product;
BEGIN
    -- wstawia nowy produkt
    product_package.insert_product(4, 'wołowina',
        '0,5 kg paczka wołowiny', 32, 10);

    -- wyświetla produkt
    product_package.display_product(4);

    -- pobiera nowy produkt i zapisuje go w v_product
    SELECT product_package.get_product(4)
    INTO v_product
    FROM dual;

    -- zmienia kilka atrybutów v_product
    v_product.description := '0,4 kg paczka wołowiny';
    v_product.price := 36;
    v_product.days_valid := 8;

    -- modyfikuje produkt
    product_package.update_product(v_product);

    -- wyświetla produkt
    product_package.display_product(4);

    -- usuwa produkt
    product_package.delete_product(4);
END product_lifecycle;
/

```

W poniższym przykładzie wywołano procedurę `product_lifecycle()`:

```

CALL product_lifecycle();
v_product.id=4
v_product.name=wołowina
v_product.description=0,5 kg paczka wołowiny
v_product.price=32
v_product.days_valid=10
sell_by_date=12/11/08

```

```
v_product.id=4
v_product.name=wołowina
v_product.description=0,4 kg paczka wołowiny
v_product.price=36
v_product.days_valid=8
sell_by_date=12/11/06
```

Procedura product_lifecycle2()

Procedura `product_lifecycle2()` wykorzystuje odwołanie obiektowe do uzyskania dostępu do produktu. Jest definiowana w następujący sposób:

```
CREATE PROCEDURE product_lifecycle2 AS
  -- deklaracja obiektu
  v_product t_product;

  -- deklaracja odwołania obiektowego
  v_product_ref REF t_product;
BEGIN
  -- wstawia nowy produkt
  product_package.insert_product(4, 'wołowina',
    '0,5 kg paczka wołowiny', 32, 10);

  -- wyświetla produkt
  product_package.display_product(4);

  -- pobiera odwołanie do nowego produktu i zapisuje je w v_product_ref
  SELECT product_package.get_product_ref(4)
  INTO v_product_ref
  FROM dual;

  -- dereferencja v_product_ref z użyciem następującego zapytania
  SELECT DEREF(v_product_ref)
  INTO v_product
  FROM dual;

  -- zmiana niektórych atrybutów v_product
  v_product.description := '0,4 kg paczka wołowiny';
  v_product.price := 36;
  v_product.days_valid := 8;

  -- aktualizacja produktu
  product_package.update_product(v_product);

  -- wyświetla produkt
  product_package.display_product(4);

  -- usuwa produkt
  product_package.delete_product(4);
END product_lifecycle2;
/
```

W tej procedurze należy zauważać, że w celu dereferencowania `v_product_ref` musimy użyć następującego zapytania:

```
SELECT DEREF(v_product_ref)
INTO v_product
FROM dual;
```

Wynika to stąd, że funkcji `DEREF()` nie można użyć bezpośrednio w kodzie PL/SQL. Na przykład powyższa instrukcja nie zostanie skompilowana w PL/SQL:

```
v_product := DEREF(v_product_ref);
```

W poniższym przykładzie wywołano procedurę `product_lifecycle2()`:

```

CALL product_lifecycle2();
v_product.id=4
v_product.name=wołowina
v_product.description=0,5 kg paczka wołowiny
v_product.price=32
v_product.days_valid=10
sell_by_date=12/11/08
v_product.id=4
v_product.name=wołowina
v_product.description=0,4 kg paczka wołowiny
v_product.price=36
v_product.days_valid=8
sell_by_date=12/11/06

```

Dziedziczenie typów

W Oracle Database 9i wprowadzono dziedziczenie typów, co pozwala na zdefiniowanie hierarchii typów obiektowych. Możemy na przykład chcieć zdefiniować reprezentujący biznesmena typ obiektowy, który będzie dziedziczył istniejące atrybuty po typie `t_person`. Typ biznesmena może rozszerzać typ `t_person` o takie atrybuty, jak zajmowane stanowisko oraz miejsce zatrudnienia. Aby było możliwe dziedziczenie po typie `t_person`, jego definicja musi zawierać klauzulę `NOT FINAL`:

```

CREATE TYPE t_person AS OBJECT (
    id INTEGER,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    dob DATE,
    phone VARCHAR2(12),
    address t_address,
    MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/

```

Klauzula `NOT FINAL` wskazuje, że definicja innego typu może dziedziczyć po `t_person`. Domyślnie definicja typu ma klauzulę `FINAL`, co oznacza, że nie można dziedziczyć po tym typie.

Poniższa instrukcja tworzy treść `t_person`. Funkcja `display_details()` zwraca `VARCHAR2` zawierający identyfikator (`id`) oraz imię i nazwisko osoby (`name`):

```

CREATE TYPE BODY t_person AS
    MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
        RETURN 'id=' || id || ', name=' || first_name || ' ' || last_name;
    END;
END;
/

```

Uruchamianie skryptu tworzącego schemat bazy danych `object_schema2`

W katalogu `SQL` znajduje się skrypt `SQL*Plus` o nazwie `object_schema2.sql`, który tworzy wszystkie elementy prezentowane w tym podrozdziale i kolejnych. Ten skrypt może zostać uruchomiony w Oracle Database 9i lub nowszych wersjach.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić `SQL*Plus`.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika `system`.
3. Uruchomić skrypt `object_schema2.sql` w `SQL*Plus` za pomocą polecenia `@`.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu C:\SQL, to należy wpisać polecenie:

```
@ C:\SQL\object_schema2.sql
```

Po zakończeniu pracy skryptu będzie zalogowany użytkownik object_user2.

Dziedziczenie atrybutów

Jeżeli chcemy, aby nowy typ dziedziczył atrybuty i metody po istniejącym, musimy użyć w jego definicji słowa kluczowego UNDER. Nasz typ reprezentujący biznesmena, który nazwiemy t_business_person, wykorzystuje słowo kluczowe UNDER w celu dziedziczenia atrybutów po t_person:

```
CREATE TYPE t_business_person UNDER t_person (
    title VARCHAR2(20),
    company VARCHAR2(20)
);
/
```

W tym przykładzie t_person jest **typem nadzawanym**, a t_business_person — **podtypem**. Typ t_business_person może zostać użyty do definiowania obiektów kolumnowych i tabel obiektowych. Poniższa instrukcja tworzy na przykład tabelę obiektową o nazwie object_business_customers:

```
CREATE TABLE object_business_customers OF t_business_person;
```

Poniższa instrukcja INSERT wstawia obiekt do tabeli object_business_customers. Na końcu konstruktora t_business_person znajdują się dwa dodatkowe atrybuty: title i company.

```
INSERT INTO object_business_customers VALUES (
    t_business_person(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
    t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
    'Kierownik', 'XYZ SA'
)
);
```

Poniższe zapytanie pobiera ten obiekt:

```
SELECT *
FROM object_business_customers
WHERE id = 1;

ID FIRST_NAME LAST_NAME DOB PHONE
----- -----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
TITLE COMPANY
-----
1 Jan Brązowy 55/02/01 800-555-1211
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345')
Kierownik XYZ SA
```

Poniższe zapytanie wywołuje funkcję display_details() tego obiektu:

```
SELECT o.display_details()
FROM object_business_customers o
WHERE id = 1;

O.DISPLAY_DETAILS()
-----
id=1, name=Jan Brązowy
```

Przy wywołaniu metody baza danych stara się ją najpierw znaleźć w podtypie. Jeżeli nie zostanie znaleziona, przeszukiwany jest typ nadzawany. Jeśli utworzono hierarchię typów, jest ona przeszukiwana w góre. Jeżeli metoda nie zostanie odnaleziona, baza danych zgłosi błąd.

Użycie podtypu zamiast typu nadzędneg o

W tym rozdziale opisano, jak można użyć podtypu zamiast typu. Oferuje to sporą swobodę przy składowaniu powiązanych typów i manipulowaniu nimi. W prezentowanych przykładach pokazano, jak użyć obiektu `t_business_person` (podtypu) zamiast obiektu `t_person` (typu nadzędneg o).

Przykłady SQL

Poniższa instrukcja tworzy tabelę `object_customers` typu `t_person`:

```
CREATE TABLE object_customers OF t_person;
```

Ta instrukcja `INSERT` wstawia obiekt typu `t_person` do tabeli (imię i nazwisko to `Jan Raczek`):

```
INSERT INTO object_customers VALUES (
    t_person(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212',
    t_address('Wolności 23', 'Gdziekolwiek', 'GDA', '12345')
)
);
```

W tej instrukcji nie ma nic nadzwyczajnego. Wstawia ona po prostu obiekt typu `t_person` do tabeli `object_customers`. Ponieważ jednak w tabeli `object_customers` są składowane obiekty typu `t_person`, a `t_person` jest typem nadzędnym dla `t_business_person`, możemy w niej składować obiekty typu `t_business_person`. Poniższa instrukcja `INSERT` wstawia na przykład informacje o kliencie (obiekt) `Stefan Czerny`:

```
INSERT INTO object_customers VALUES (
    t_business_person(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
    t_address('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'),
    'Kierownik', 'XYZ SA'
)
);
```

Tabela `object_customers` zawiera teraz dwa obiekty: wcześniej dodany `t_person` (`Jan Raczek`) oraz nowy `t_business_person` (`Stefan Czerny`). Poniższe zapytanie pobiera je. W wynikach brakuje atrybutów `title` i `company` obiektu `Stefan Czerny`:

```
SELECT *
FROM object_customers o;
-----  

ID FIRST_NAME LAST_NAME DOB PHONE  

-----  

ADDRESS(STREET, CITY, STATE, ZIP)  

-----  

1 Jan Raczek 65/05/03 800-555-1212  

T_ADDRESS('Wolności 23', 'Gdziekolwiek', 'GDA', '12345')  

-----  

2 Stefan Czerny 55/03/03 800-555-1212  

T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345')
```

Pełny zestaw atrybutów obiektu `Stefan Czerny` możemy zobaczyć, używając w zapytaniu funkcji `VALUE()`, co obrazuje kolejny przykład. Należy zauważyc, że obiekty `Jan Raczek` i `Stefan Czerny` są różnego typu (odpowiednio: `t_person` i `t_business_person`) oraz że w wynikach pojawiły się atrybuty `title` i `company` obiektu `Stefan Czerny`:

```
SELECT VALUE (o)
FROM object_customers o;
-----  

VALUE(o)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP)  

-----  

T_PERSON(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212', T_ADDRESS('Wolności 23',
'Gdziekolwiek', 'GDA', '12345'))  

-----  

T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212', T_ADDRESS('Rynek 2',
'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Przykłady PL/SQL

Praca z typami i podtypami jest również możliwa w PL/SQL. Na przykład poniższa procedura, `subtypes_and_supertypes()`, pracuje z obiektami typu `t_business_person` i `t_person`:

```

CREATE PROCEDURE subtypes_and_supertypes AS
    -- tworzenie obiektów
    v_business_person t_business_person :=
        t_business_person(
            1, 'Jan', 'Raczek',
            '55/02/01', '800-555-1211',
            t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
            'Kierownik', 'XYZ SA'
        );
    v_person t_person :=
        t_person(1, 'Jan', 'Raczek', '55/02/01', '800-555-1211',
            t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'));
    v_business_person2 t_business_person;
    v_person2 t_person;
BEGIN
    -- przypisuje v_business_person do v_person2
    v_person2 := v_business_person;
    DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
    DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
        v_person2.first_name);
    DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
        v_person2.last_name);

    -- poniższe wiersze nie zostaną skompilowane, ponieważ v_person2
    -- jest typu t_person, a t_person nie zna
    -- dodatkowych atrybutów title i company
    -- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
    -- v_person2.title);
    -- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
    -- v_person2.company);

    -- poniższy wiersz nie zostanie skompilowany,
    -- nie można bezpośrednio przypisać obiektu
    -- t_person do obiektu t_business_person
    -- v_business_person2 = v_person;
END subtypes_and_supertypes;
/

```

Poniższy przykład prezentuje wynik wywołania procedury `subtypes_and_supertypes()`:

```

SET SERVEROUTPUT ON
CALL subtypes_and_supertypes();
v_person2.id = 1
v_person2.first_name = Jan
v_person2.last_name = Raczek

```

Obiekty NOT SUBSTITUTABLE

Jeżeli chcemy zapobiec użyciu podtypu zamiast obiektu typu nadziednego, możemy oznaczyć tabelę lub kolumnę obiektową jako **NOT SUBSTITUTABLE**. Na przykład poniższa instrukcja tworzy tabelę `object_customers2`:

```

CREATE TABLE object_customers_not_subs OF t_person
NOT SUBSTITUTABLE AT ALL LEVELS;

```

Klauzula `NOT SUBSTITUTABLE AT ALL LEVELS` oznacza, że do tabeli można wstawiać tylko obiekty typu `t_person`. Jeżeli spróbujemy wstawić do niej obiekt typu `t_business_person`, zostanie zgłoszony błąd:

```
SQL> INSERT INTO object_customers_not_subs VALUES (
  2   t_business_person(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
  3   t_address('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'),
  4   'Kierownik', 'XYZ SA'
  5 )
  6 );
t_business_person(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
*
BŁĄD w linii 2:
ORA-00932: niespójne typy danych: oczekiwano OBJECT_USER2.T_PERSON, uzyskano
OBJECT_USER2.T_BUSINESS_PERSON
```

W ten sposób można również oznaczyć kolumnę obiektową. Poniższa instrukcja tworzy na przykład tabelę z kolumną obiektową `product`, w której mogą być składowane jedynie obiekty typu `t_product`:

```
CREATE TABLE products (
  product t_product,
  quantity_in_stock INTEGER
)
COLUMN product NOT SUBSTITUTABLE AT ALL LEVELS;
```

Próba wstawienia do kolumny `product` obiektu o innym typie niż `t_product` spowoduje zgłoszenie błędu.

Inne przydatne funkcje obiektów

W poprzednich podrozdziałach opisano zastosowanie funkcji `REF()`, `DEREF()` i `VALUE()`. W tym podroziale zostaną przedstawione kolejne funkcje pracujące z obiektami:

- **`IS OF()`** sprawdza, czy obiekt jest podanego typu lub podtypu,
- **`TREAT()`** sprawdza w trakcie wykonywania, czy typ obiektu może być traktowany jako typ nadzędny,
- **`SYS_TYPEID()`** zwraca identyfikator typu obiektu.

Funkcja `IS OF()`

Funkcja `IS OF()` służy do sprawdzania, czy obiekt jest określonego typu lub podtypu. W poniższym zapytaniu użyto na przykład funkcji `IS OF()` do sprawdzenia, czy obiekty w tabeli `object_customers` są typu `t_business`, a ponieważ są, zapytanie zwraca wiersz:

```
SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF(t_business_person);

-----  
VALUE(o)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----  
T_BUSINESS_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Za pomocą funkcji `IS OF()` możemy również sprawdzić, czy obiekt jest podtypem określonego typu. Na przykład obiekty w tabeli `object_business_customers` są typu `t_business_person`, który jest podtypem `t_person`, dlatego też to zapytanie zwraca taki sam wynik jak poprzednie:

```
SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF(t_person);

-----  
VALUE(o)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----  
T_BUSINESS_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Do funkcji `IS OF()` możemy przesyłać kilka typów. Na przykład:

```

SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF (t_business_person, t_person);

VALUE(O)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_BUSINESS_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')

```

We wcześniejszym podrozdziale, zatytułowanym „Użycie podtypu zamiast typu nadziedznego”, wstawiliśmy do tabeli `object_customers` obiekt typu `t_person` (`Jan Raczek`) oraz `t_business_person` (`Stefan Czerny`). Poniższe zapytanie zwraca je:

```

SELECT VALUE(o)
FROM object_customers o;

VALUE(O)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212',
T_ADDRESS('Wolności 23', 'Gdziekolwiek', 'GDA', '12345'))

T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')

```

Ponieważ `t_business_person` jest podtypem `t_person`, `IS OF (t_person)` zwróci `true`, jeżeli będzie sprawdzany obiekt typu `t_business_person` lub `t_person`. Obrazuje to poniższe zapytanie zwracające informacje zarówno z obiektu `Jan Raczek`, jak i `Stefan Czerny`:

```

SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (t_person);

VALUE(O)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212',
T_ADDRESS('Wolności 23', 'Gdziekolwiek', 'GDA', '12345'))

T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')

```

Łącząc słowo kluczowe `ONLY` z funkcją `IS OF()`, możemy sprawdzić obiekty tylko określonego typu: funkcja `IS OF()` zwróci `false` dla obiektów innego typu w hierarchii. Na przykład `IS OF (ONLY t_person)` zwróci `true` dla obiektów typu `t_person` i `false` dla obiektów typu `t_business_person`. W ten sposób możemy użyć `IS OF (ONLY t_person)` do ograniczenia wyników zwracanych przez zapytanie tabeli `object_customers` jedynie obiektu `Jan Raczek`, co obrazuje poniższy przykład:

```

SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_person);

VALUE(O)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212',
T_ADDRESS('Wolności 23', 'Gdziekolwiek', 'GDA', '12345'))

```

`IS OF (ONLY t_business_person)` również zwróci `true` dla obiektów typu `t_business_person` i `false` dla obiektów typu `t_person`. Na przykład poniższe zapytanie pobiera jedynie obiekty typu `t_business_person`, są więc zwracane tylko informacje z obiektu `Stefan Czerny`:

```

SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_business_person);

VALUE(O)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
```

```
T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Po słowie kluczowym **ONLY** możemy umieścić kilka typów. Na przykład **IS OF (ONLY t_person, t_business_person)** zwraca **true** jedynie dla obiektów typu **t_person** i **t_business_person**. Obrzucaje to poniższe zapytanie, zwracając zgodnie z oczekiwaniemi informacje z obiektów Jan Raczek i Stefan Czerny:

```
SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_person, t_business_person);

VALUE(o)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jan', 'Raczek', '65/05/03', '800-555-1212',
T_ADDRESS('Wolności 23', 'Gdziekolwiek', 'GDA', '12345'))

T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Funkcja **IS OF()** może również zostać użyta w PL/SQL. Na przykład poniższa procedura **check_types()** tworzy obiekty typu **t_business_person** oraz **t_person** i sprawdza ich typy za pomocą funkcji **IS OF()**:

```
CREATE PROCEDURE check_types AS
-- tworzenie obiektów
v_business_person t_business_person :=
    t_business_person(
        1, 'Jan', 'Raczek',
        '55/02/01', '800-555-1211',
        t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
        'Manager', 'XYZ Corp'
    );
v_person t_person :=
    t_person(1, 'Jan', 'Raczek', '55/02/01', '800-555-1211',
        t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'));
BEGIN
-- sprawdzanie typów obiektów
IF v_business_person IS OF (t_business_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_business_person jest typu ' ||
        't_business_person');
END IF;
IF v_person IS OF (t_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_person jest typu t_person');
END IF;
IF v_business_person IS OF (t_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_business_person jest typu t_person');
END IF;
IF v_business_person IS OF (t_business_person, t_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_business_person jest ' ||
        'typu t_business_person lub t_person');
END IF;
IF v_business_person IS OF (ONLY t_business_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_business_person jest tylko ' ||
        'typu t_business_person');
END IF;
IF v_business_person IS OF (ONLY t_person) THEN
    DBMS_OUTPUT.PUT_LINE('v_business_person jest tylko ' ||
        'typu t_person');
ELSE
    DBMS_OUTPUT.PUT_LINE('v_business_person nie jest tylko ' ||
        'typu t_person');
END IF;
END check_types;
/
```

Poniżej przedstawiono wynik wywołania procedury **check_types()**:

```
SET SERVEROUTPUT ON
CALL check_types();
v_business_person jest typu t_business_person
v_person jest typu t_person
v_business_person jest typu t_person
v_business_person jest typu t_business_person lub t_person
v_business_person jest tylko typu t_business_person
v_business_person nie jest tylko typu t_person
```

Funkcja TREAT()

Funkcja TREAT() pozwala sprawdzić w trakcie wykonywania, czy obiekt podtypu może być traktowany jako obiekt typu nadzędnego. Jeżeli tak, funkcja TREAT() zwraca ten obiekt, a w przeciwnym razie zwraca NULL. Na przykład ponieważ t_business_person jest podtypem t_person, obiekt t_business_person może być traktowany jako obiekt t_person. Zaprezentowano to już wcześniej w podrózdziele „Użycie podtypu zamiast typu nadzędnegiego”: obiekt typu t_business_person (Stefan Czerny) był wstawiany do tabeli object_customers, w której normalnie są składowane obiekty typu t_person.

W poniższym zapytaniu użyto funkcji TREAT() do sprawdzenia, czy obiekt Stefan Czerny może być traktowany jako obiekt t_person:

```
SELECT NVL2(TREAT(VALUE(o) AS t_person), 'tak', 'nie')
FROM object_customers o
WHERE first_name = 'Stefan' AND last_name = 'Czerny';
```

```
NVL
---
tak
```

Funkcja NVL2() zwraca tak, ponieważ TREAT(VALUE(o) AS t_person) zwraca obiekt (czyli wartość inną niż NULL). To oznacza, że obiekt Stefan Czerny może być traktowany jako obiekt typu t_person.

Kolejne zapytanie sprawdza, czy obiekt Jan Raczek (typu t_person) może być traktowany jako obiekt t_business_person, a ponieważ nie może, funkcja TREAT() zwraca NULL, NVL2() zwraca nie:

```
SELECT NVL2(TREAT(VALUE(o) AS t_business_person), 'tak', 'nie')
FROM object_customers o
WHERE first_name = 'Jan' AND last_name = 'Raczek';
```

```
NVL
---
nie
```

Ponieważ funkcja TREAT() zwraca NULL dla całego obiektu, wszystkie atrybuty obiektu również mają wartość NULL. Na przykład poniższe zapytanie sięga po atrybut first_name obiektu Jan Raczek — zgodnie z oczekiwaniemi jest zwracana wartość NULL:

```
SELECT
  NVL2(TREAT(VALUE(o) AS t_business_person).first_name, 'nie null', 'null')
FROM object_customers o
WHERE first_name = 'Jan' AND last_name = 'Raczek';

NVL2
-----
null
```

W kolejnym zapytaniu użyto funkcji TREAT() do sprawdzenia, czy obiekt Jan Raczek może być traktowany jako obiekt typu t_person, a ponieważ to jest obiekt t_person, jest zwracane tak:

```
SELECT NVL2(TREAT(VALUE(o) AS t_person).first_name, 'tak', 'nie')
FROM object_customers o
WHERE first_name = 'Jan' AND last_name = 'Raczek';

NVL
---
tak
```

376 Oracle Database 12c i SQL. Programowanie

Za pomocą funkcji TREAT() można również pobrać obiekt. Na przykład poniższe zapytanie pobiera obiekt Stefan Czerny:

```
SELECT TREAT(VALUE(o) AS t_business_person)
FROM object_customers o
WHERE first_name = 'Stefan' AND last_name = 'Czerny';

TREAT(VALUE(o)AS_T_BUSINESS_PERSON)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS
-----
T_BUSINESS_PERSON(2, 'Stefan', 'Czerny', '55/03/03', '800-555-1212',
T_ADDRESS('Rynek 2', 'Gdziekolwiek', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Jeżeli w tym zapytaniu spróbujemy pobrać obiekt Jan Raczek, zgodnie z oczekiwaniami zostanie zwrocona wartość NULL. W związku z tym w wynikach poniższego zapytania nie pojawi się nic:

```
SELECT TREAT(VALUE(o) AS t_business_person)
FROM object_customers o
WHERE first_name = 'Jan' AND last_name = 'Raczek';

TREAT(VALUE(o)AS_T_BUSINESS_PERSON)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS
-----
```

Sprawdźmy, jak możemy użyć funkcji TREAT() z tabelą object_business_customers, która zawiera obiekt typu t_business_person (Jan Brązowy):

```
SELECT VALUE(o)
FROM object_business_customers o;

VALUE(o)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP
-----
T_BUSINESS_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

W poniższym zapytaniu użyto funkcji TREAT() do sprawdzenia, czy obiekt Jan Brązowy może być traktowany jako obiekt t_person, a ponieważ może być tak traktowany (typ t_business_person jest podtypem t_person), zapytanie zwraca tak:

```
SELECT NVL2(TREAT(VALUE(o) AS t_person), 'tak', 'nie')
FROM object_business_customers o
WHERE first_name = 'Jan' AND last_name = 'Brązowy';
```

```
NVL
---
tak
```

Poniższy przykład przedstawia obiekt zwracany przez funkcję TREAT() w zapytaniu tabeli object_business_customers. Należy zauważyć, że zwracane są atrybuty title i company obiektu Jan Brązowy:

```
SELECT TREAT(VALUE(o) AS t_person)
FROM object_business_customers o;

TREAT(VALUE(o)AS_T_PERSON)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET,
-----
T_BUSINESS_PERSON(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'), 'Kierownik', 'XYZ SA')
```

Funkcja TREAT() może być również używana w PL/SQL. Poniższa procedura o nazwie treat_example() obrazuje użycie tej funkcji (dokładny opis działania funkcji TREAT() w PL/SQL znajduje się w komentariuszach do kodu tej procedury):

```
CREATE PROCEDURE treat_example AS
-- tworzenie obiektów
v_business_person t_business_person := 
t_business_person(
 1, 'Jan', 'Brązowy',
 '55/02/01', '800-555-1211',
 t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
```

```

        'Kierownik', 'XYZ SA'
);
v_person t_person :=
    t_person(1, 'Jan', 'Brązowy', '55/02/01', '800-555-1211',
    t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'));
v_business_person2 t_business_person;
v_person2 t_person;
BEGIN
-- przypisuje v_business_person v_person2
    v_person2 := v_business_person;
    DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
    DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
        v_person2.first_name);
    DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
        v_person2.last_name);

-- poniższe wiersze nie zostaną skompilowane, ponieważ v_person2
-- jest typu t_person, a t_person nie posiada
-- dodatkowych atrybutów title i company
-- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
-- v_person2.title);
-- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
-- v_person2.company);

-- użycie TREAT przy przypisywaniu v_business_person do v_person2
DBMS_OUTPUT.PUT_LINE('Użycie TREAT');
    v_person2 := TREAT(v_business_person AS t_person);
    DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
    DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
        v_person2.first_name);
    DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
        v_person2.last_name);

-- poniższe wiersze wciąż nie będą komplikowane, ponieważ v_person2
-- jest typu t_person, a t_person nie posiada
-- dodatkowych atrybutów title i company
-- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
-- v_person2.title);
-- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
-- v_person2.company);

-- poniższe wiersze zostaną skompilowane, ponieważ użyto TREAT
DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
    TREAT(v_person2 AS t_business_person).title);
DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
    TREAT(v_person2 AS t_business_person).company);

-- poniższy wiersz nie zostanie skompilowany, ponieważ nie można
-- bezpośrednio przypisać obiektu t_person do obiektu t_business_person
-- v_business_person2 = v_person;

-- poniższy wiersz powoduje wystąpienie błędu w trakcie wykonywania,
-- ponieważ nie można przypisać obiektu typu nadziednego (v_person)
-- do obiektu podtypu (v_business_person)
-- v_business_person2 = TREAT(v_person AS t_business_person);
END treat_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `treat_example()`:

```

SET SERVEROUTPUT ON
CALL treat_example();
v_person2.id = 1
v_person2.first_name = Jan

```

```
v_person2.last_name = Brązowy
Użycie TREAT
v_person2.id = 1
v_person2.first_name = Jan
v_person2.last_name = Brązowy
v_person2.title = Kierownik
v_person2.company = XYZ SA
```

Funkcja SYS_TYPEID()

Funkcja `SYS_TYPEID()` zwraca identyfikator typu obiektu. Na przykład w poniższym zapytaniu użyto jej do pobrania identyfikatora typu obiektu w tabeli `object_business_customers`:

```
SELECT first_name, last_name, SYS_TYPEID(VALUE(o))
FROM object_business_customers o;
```

FIRST_NAME	LAST_NAME	SY
Jan	Brązowy	02

Szczegółowe informacje o typach zdefiniowanych przez użytkownika można uzyskać za pośrednictwem perspektywy `user_types`. Poniższe zapytanie pobiera szczegóły typu o identyfikatorze (`typeid`) równym 02 (został on zwrócony przez `SYS_TYPEID()` w poprzednim przykładzie), którego `type_name` ma wartość `T_BUSINESS_PERSON`:

```
SELECT typecode, attributes, methods, supertype_name
FROM user_types
WHERE typeid = '02'
AND type_name = 'T_BUSINESS_PERSON';
```

TYPECODE	ATTRIBUTES	METHODS	SUPERTYPE_NAME
OBJECT	8	1	T_PERSON

W wynikach tego zapytania widać, że typem nadzawanym dla `t_business_person` jest `t_person`. `t_business_person` posiada ponadto osiem atrybutów i jedną metodę.

Typy obiektowe NOT INSTANTIABLE

Typ obiektowy możemy oznać jako `NOT INSTANTIABLE`, co zapobiega tworzeniu obiektów tego typu. Ta możliwość przynosi się, gdy używamy typu jedynie jako abstrakcyjnego typu nadzawanego i nigdy nie tworzymy jego obiektów.

Możemy na przykład utworzyć typ abstrakcyjny `t_vehicle` (pojazd) i użyć go jako typu nadzawanego wobec podtypów `t_car` (samochód) i `t_motorcycle`. Wówczas będą tworzone jedynie obiekty typu `t_car` i `t_motorcycle`, ale nie `t_vehicle`.

Poniższa instrukcja tworzy typ `t_vehicle`, oznaczony jako `NOT INSTANTIABLE`:

```
CREATE TYPE t_vehicle AS OBJECT (
    id INTEGER,
    make VARCHAR2(15),
    model VARCHAR2(15)
) NOT FINAL NOT INSTANTIABLE;
/
```



Typ `t_vehicle` jest również oznaczony jako `NOT FINAL`, ponieważ typ `NOT INSTANTIABLE` nie może być oznaczony jako `FINAL` — w takim przypadku nie mógłby zostać użyty jako typ nadzawany, a przecież takie jest jego główne przeznaczenie.

Kolejny przykład tworzy podtyp `t_car` typu nadzawanego `t_vehicle`. Należy zauważać, że `t_car` ma dodatkowy atrybut `convertible`, który będzie używany do określenia, czy samochód jest kabrioletem (t), czy też nie (n):

```
CREATE TYPE t_car UNDER t_vehicle (
    convertible CHAR(1)
);
/

```

Poniższy przykład tworzy podtyp `t_motorcycle` typu nadzecznego `t_vehicle`. Należy zauważyć, że `t_motorcycle` posiada dodatkowy atrybut `sidecar`, określający, czy motocykl jest wyposażony w boczną przyczepę (`t` lub `n`):

```
CREATE TYPE t_motorcycle UNDER t_vehicle (
    sidecar CHAR(1)
);
/

```

Kolejny przykład tworzy tabele `vehicles`, `cars` i `motorcycles`, które są tabelami obiektowymi o typach (odpowiednio): `t_vehicle`, `t_car` it `t_motorcycle`:

```
CREATE TABLE vehicles OF t_vehicle;
CREATE TABLE cars OF t_car;
CREATE TABLE motorcycles OF t_motorcycle;
```

Ponieważ typ `t_vehicle` jest oznaczony jako `NOT INSTANTIABLE`, nie można wstawić obiektu do tabeli `vehicles`. Jeżeli spróbujemy, zostanie zwrócony błąd:

```
SQL> INSERT INTO vehicles VALUES (
  2     t_vehicle(1, 'Toyota', 'MR2', '55/02/01')
  3 );
t_vehicle(1, 'Toyota', 'MR2', '55/02/01')
*
```

BŁĄD w linii 2:
ORA-22826: nie można skonstruować instancji z typu NOT INSTANTIABLE

Poniższy przykład wstawia obiekty do tabel `cars` i `motorcycles`:

```
INSERT INTO cars VALUES (
    t_car(1, 'Toyota', 'MR2', 'Y')
);

INSERT INTO motorcycles VALUES (
    t_motorcycle (1, 'Harley-Davidson', 'V-Rod', 'N')
);
```

Poniższe zapytania pobierają obiekty z tabel `cars` i `motorcycles`:

```
SELECT *
FROM cars;
```

ID	MAKE	MODEL	C
1	Toyota	MR2	Y

```
SELECT *
FROM motorcycles;
```

ID	MAKE	MODEL	S
1	Harley-Davidson	V-Rod	N

Konstruktory definiowane przez użytkownika

W PL/SQL możemy definiować własne konstruktory inicjalizujące nowy obiekt. Za ich pomocą możemy na przykład programowo przypisywać atrybutom nowego obiektu wartości domyślne.

Poniższy przykład tworzy typ o nazwie `t_person2`, który deklaruje dwa konstruktory o różnej liczbie parametrów:

```

CREATE OR REPLACE TYPE t_person2 AS OBJECT (
    id          INTEGER,
    first_name VARCHAR2(10),
    last_name  VARCHAR2(10),
    dob         DATE,
    phone       VARCHAR2(12),
    CONSTRUCTOR FUNCTION t_person2(
        p_id          INTEGER,
        p_first_name VARCHAR2,
        p_last_name  VARCHAR2
    ) RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION t_person2(
        p_id          NUMBER,
        p_first_name VARCHAR2,
        p_last_name  VARCHAR2,
        p_dob         DATE
    ) RETURN SELF AS RESULT
);
/

```

W deklaracjach konstruktorów:

- Do określenia konstruktorów użyto słów kluczowych **CONSTRUCTOR FUNCTION**.
- Słowa kluczowe **RETURN SELF AS RESULT** wskazują, że bieżący przetwarzany obiekt jest zwracany przez każdy konstruktor; **SELF** reprezentuje aktualnie przetwarzany obiekt. Oznacza to, że konstruktor zwraca utworzony przez siebie obiekt.
- Pierwszy konstruktor przyjmuje trzy parametry (**p_id**, **p_first_name** i **p_last_name**), a drugi konstruktor przyjmuje cztery parametry (**p_id**, **p_first_name**, **p_last_name** i **p_dob**).

Deklaracje konstruktorów nie zawierają faktycznych definicji kodu konstruktorów. Definicje są umieszczone w treści typu, który jest tworzony przez poniższą instrukcję:

```

CREATE TYPE BODY t_person2 AS
    CONSTRUCTOR FUNCTION t_person2(
        p_id          INTEGER,
        p_first_name VARCHAR2,
        p_last_name  VARCHAR2
    ) RETURN SELF AS RESULT IS
        BEGIN
            SELF.id := p_id;
            SELF.first_name := p_first_name;
            SELF.last_name := p_last_name;
            SELF.dob := SYSDATE;
            SELF.phone := '555-1212';
            RETURN;
        END;
    CONSTRUCTOR FUNCTION t_person2(
        p_id          INTEGER,
        p_first_name VARCHAR2,
        p_last_name  VARCHAR2,
        p_dob         DATE
    ) RETURN SELF AS RESULT IS
        BEGIN
            SELF.id := p_id;
            SELF.first_name := p_first_name;
            SELF.last_name := p_last_name;
            SELF.dob := p_dob;
            SELF.phone := '555-1213';
            RETURN;
        END;
    END;
/

```

Należy zauważyć, że:

- W konstruktach użyto słowa kluczowego SELF w celu odwołania się do tworzonego obiektu. Na przykład `SELF.id := p_id` ustawia wartość atrybutu `id` nowego obiektu na wartość parametru `p_id` przesyłanego do konstruktora.
- Pierwszy konstruktor przypisuje atrybutom `id`, `first_name` i `last_name` wartości parametrów `p_id`, `p_first_name` i `p_last_name` przesyłanych do konstruktora. Atrybutowi `dob` jest przypisywana bieżąca data i godzina zwracane przez funkcję `SYSDATE()`, a atrybutowi `phone` — wartość `555-1212`.
- Drugi konstruktor przypisuje atrybutom `id`, `first_name`, `last_name` i `dob` wartości parametrów `p_id`, `p_first_name`, `p_last_name` i `p_dob` przesyłanych do konstruktora; atrybutowi `phone` jest przypisywana wartość `555-1213`.

Choć nie zostało to przedstawione, baza danych automatycznie dostarcza domyślny konstruktor, który przyjmuje pięć parametrów i przypisuje każdemu atrybutowi wartość odpowiedniego parametru przesłanego do konstruktora. Wkrótce zostanie przedstawiony odpowiedni przykład.



Uwaga

Konstruktory stanowią przykład **przeciążania metod**, czyli sytuacji, w której w tym samym typie są definiowane metody o tej samej nazwie, lecz różnych parametrach. Metoda może zostać przeciążona przez dostarczenie innej liczby parametrów, innego typu parametrów lub zmianę kolejności parametrów.

Poniższy przykład stanowi opis typu `t_person2`. W wynikach znajdują się definicje konstruktorów:

```
DESCRIBE t_person2
Name          NULL?    Type
-----        -----
ID           NUMBER(38)
FIRST_NAME   VARCHAR2(10)
LAST_NAME    VARCHAR2(10)
DOB          DATE
PHONE        VARCHAR2(12)

METHOD
-----
FINAL CONSTRUCTOR FUNCTION T_PERSON2 RETURNS SELF AS RESULT
Argument Name      Type      In/Out Default?
-----            -----
P_ID             NUMBER    IN
P_FIRST_NAME    VARCHAR2  IN
P_LAST_NAME     VARCHAR2  IN

METHOD
-----
FINAL CONSTRUCTOR FUNCTION T_PERSON2 RETURNS SELF AS RESULT
Argument Name      Type      In/Out Default?
-----            -----
P_ID             NUMBER    IN
P_FIRST_NAME    VARCHAR2  IN
P_LAST_NAME     VARCHAR2  IN
P_DOB            DATE      IN
```

Poniższa instrukcja tworzy tabelę o typie `t_person2`:

```
CREATE TABLE object_customers2 OF t_person2;
```

Poniższa instrukcja `INSERT` wstawia obiekt do tej tabeli. Należy zauważyć, że do konstruktora `t_person2` są przesyłane trzy parametry:

```
INSERT INTO object_customers2 VALUES (
  t_person2(1, 'Jerzy', 'Kowalski')
);
```

Ponieważ do `t_person2` przesyłane są trzy parametry, powyższa instrukcja `INSERT` wykorzystuje pierwszy konstruktor. Przypisuje on atrybutom `id`, `first_name` i `last_name` wartości 1, Jerzy i Kowalski. Pozostałym atrybutom, `dob` i `phone`, jest przypisywana (odpowiednio) wartość zwracana przez funkcję `SYSDATE()` oraz literal 555-1212.

Poniższe zapytanie pobiera nowy obiekt:

```
SELECT *
FROM object_customers2
WHERE id = 1;

ID FIRST_NAME LAST_NAME DOB PHONE
-----
1 Jerzy Kowalski 08/10/30 555-1212
```

Kolejna instrukcja `INSERT` wstawia do tabeli inny obiekt. Należy zauważyć, że do konstruktora `t_person2` jest przesyłanych pięć parametrów:

```
INSERT INTO object_customers2 VALUES (
    t_person2(2, 'Grzegorz', 'Nowak', '65/04/03')
);
```

Ponieważ do konstruktora `t_person2` przesyłane są cztery parametry, powyższa instrukcja wykorzystuje drugi konstruktor. Przypisuje on atrybutom `id`, `first_name`, `last_name` i `dob` wartości (odpowiednio) 2, Grzegorz, Nowak i 65/04/03; pozostałemu atrybutowi `phone` jest przypisywana wartość 555-1213.

Poniższe zapytanie pobiera nowy obiekt:

```
SELECT *
FROM object_customers2
WHERE id = 2;

ID FIRST_NAME LAST_NAME DOB PHONE
-----
2 Grzegorz Nowak 65/04/03 555-1213
```

Poniższa instrukcja `INSERT` wstawia kolejny obiekt do tabeli. Do konstruktora `t_person2` jest przesyłanych pięć parametrów:

```
INSERT INTO object_customers2 VALUES (
    t_person2(3, 'Antoni', 'Leszcz', '75/06/05', '555-1214')
);
```

Ponieważ do konstruktora `t_person2` jest przesyłanych pięć parametrów, powyższa instrukcja `INSERT` wykorzystuje domyślny konstruktor. Przypisuje on atrybutom `id`, `first_name`, `last_name`, `dob` i `phone` wartości (odpowiednio): 3, Antoni, Leszcz, 75/06/05 i 555-1214.

Poniższe zapytanie pobiera nowy obiekt:

```
SELECT *
FROM object_customers2
WHERE id = 3;

ID FIRST_NAME LAST_NAME DOB PHONE
-----
3 Antoni Leszcz 75/06/05 555-1214
```

Przesłanianie metod

Tworząc podtyp typu nadzawanego, możemy przesłonić metodę z typu nadzawanego metodą zdefiniowaną w podtypie. Uzyskujemy dzięki temu znaczny stopień elastyczności podczas definiowania metod w hierarchii typów.

Poniższe instrukcje tworzą typ nadzawny o nazwie `t_person3`. Funkcja `display_details()` zwraca typ `VARCHAR2`, zawierający wartości atrybutów obiektu:

```
CREATE TYPE t_person3 AS OBJECT (
    id INTEGER,
```

```

first_name VARCHAR2(10),
last_name VARCHAR2(10),
MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/
CREATE TYPE BODY t_person3 AS
  MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
      RETURN 'id=' || id ||
        ', name=' || first_name || ' ' || last_name;
    END;
END;
/

```

Kolejny zestaw instrukcji tworzy typ `t_business_person3` jako podtyp `t_person3`. Funkcja `display_details()` została przesłonięta za pomocą słowa kluczowego `OVERRIDING` i zwraca typ `VARCHAR2`, zawierający pierwotne i rozszerzone wartości atrybutów obiektu:

```

CREATE TYPE t_business_person3 UNDER t_person3 (
  title VARCHAR2(20),
  company VARCHAR2(20),
  OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2
);
/
CREATE TYPE BODY t_business_person3 AS
  OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
      RETURN 'id=' || id ||
        ', name=' || first_name || ' ' || last_name ||
        ', title=' || title || ', company=' || company;
    END;
END;
/

```

Użycie słowa kluczowego `OVERRIDING` wskazuje, że funkcja `display_details()` w `t_business_person3` przesyła funkcję `display_details()` zdefiniowaną w `t_person3`. Tym samym wywołanie `display_details()` z `t_business_person3` wywoła `display_details()` z `t_business_person3`, a nie funkcję `display_details()` z typu nadzawanego `t_person3`.



W kolejnym podrozdziale opisano, jak można wywołać metodę zdefiniowaną w typie nadzewanym bezpośrednio z poziomu podtypu. To pozwala uniknąć powtarzania w podtypie kodu znajdującego się już w typie nadzewanym. Takie bezpośrednie wywoływanie jest możliwe dzięki **ogólnionemu wywoływaniu** (ang. *generalized invocation*).

Poniższe instrukcje tworzą tabelę o nazwie `object_business_customers3` i wstawiają do niej obiekt:

```
CREATE TABLE object_business_customers3 OF t_business_person3;
```

```
INSERT INTO object_business_customers3 VALUES (
t_business_person3(1, 'Jan', 'Brązowy', 'Kierownik', 'XYZ SA')
);
```

Poniższy przykład wywołuje funkcję `display_details()` z użyciem `object_business_customers3`:

```
SELECT o.display_details()
FROM object_business_customers3 o
WHERE id = 1;
O.DISPLAY_DETAILS()
-----
id=1, name=Jan Brązowy, title=Kierownik, company=XYZ SA
```

Ponieważ jest wywoływana funkcja `display_details()` zdefiniowana w `t_business_person3`, zwróci ona typ `VARCHAR2` zawierającą atrybuty `id`, `first_name` i `last_name`, a także `title` i `company`.

Uogólnione wywoływanie

Jak pokazano w poprzednim podrozdziale, możliwe jest przesłonięcie metody zdefiniowanej w typie nadzewnętrznym metodą zdefiniowaną w typie podrzędnym. **Uogólnione wywoływanie** zostało wprowadzone w Oracle Database 11g i umożliwia wywoływanie z poziomu podtypu metody zdefiniowanej w typie nadzewnętrznym. Jak się przekonamy, pozwala to uniknąć powtarzania w podtypie kodu znajdującego się już w typie nadzewnętrznym.

Uruchomienie skryptu tworzącego schemat bazy danych object_schema3

W katalogu SQL znajduje się skrypt o nazwie *object_schema3.sql*, który tworzy użytkownika **object_user3** z hasłem **object_password3**. Może on zostać uruchomiony w Oracle Database 11g lub nowszej.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika **system**.
3. Uruchomić skrypt *object_schema3.sql* w SQL*Plus za pomocą polecenia @.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu C:\SQL, to należy wpisać polecenie:

```
@ C:\SQL\object_schema3.sql
```

Po zakończeniu jego pracy będzie zalogowany użytkownik **object_user3**.

Dziedziczenie atrybutów

Poniższe instrukcje tworzą typ nadzędny **t_person**. Funkcja **display_details()** zwraca typ **VARCHAR2**, zawierający wartości atrybutów:

```
CREATE TYPE t_person AS OBJECT (
    id INTEGER,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/
```

```
CREATE TYPE BODY t_person AS
    MEMBER FUNCTION display_details RETURN VARCHAR2 IS
        BEGIN
            RETURN 'id=' || id ||
                   ', name=' || first_name || ' ' || last_name;
        END;
    END;
/
```

Kolejny zestaw instrukcji tworzy typ **t_business_person** będący podtypem **t_person**. Funkcja **display_details()** została przesłonięta za pomocą słowa kluczowego **OVERRIDING**:

```
CREATE TYPE t_business_person UNDER t_person (
    title VARCHAR2(20),
    company VARCHAR2(20),
    OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2
);
/
```

```

CREATE TYPE BODY t_business_person AS
  OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2 IS
  BEGIN
    -- użycie uogólnionego wywołania w celu wywołania display_details()
    -- z t_person
    RETURN (SELF AS t_person).display_details ||
      ', title=' || title || ', company=' || company;
  END;
END;
/

```

Funkcja `display_details()` w definicji `t_business_person` przesyła funkcję `display_details()` zdefiniowaną w `t_person`. Poniższy wiersz z definicji funkcji `display_details()` wykorzystuje uogólnione wywołanie w celu wywołania z poziomu podtypu metody zdefiniowanej w typie nadziedzonym:

```

  RETURN (SELF AS t_person).display_details ||
    ', title=' || title || ', company=' || company;

```

Wyrażenie `(SELF AS t_person).display_details` powoduje potraktowanie obiektu bieżącego typu (czyli `t_business_person`) jako obiektu typu `t_person` i wywołanie funkcji `display_details()` z `t_person`. Wywołanie `display_details()` z `t_business_person` powoduje więc wywołanie funkcji `display_details()` z `t_person` (co wyświetla wartości atrybutów `id`, `first_name` i `last_name`), a następnie wyświetlenie wartości atrybutów `title` i `company`. To oznacza, że w definicji funkcji `t_business_person.display_details()` nie było konieczne ponowne pisanie kodu znajdującego się w `t_person.display_details()`, co pozwala zaoszczędzić nieco czasu. Jeżeli w definicjach typów występują bardziej złożone metody, ta właściwość pozwala zaoszczędzić sporo wysiłku, a także ułatwia późniejsze zarządzanie kodem.

Poniższe instrukcje tworzą tabelę o nazwie `object_business_customers` i wstawiają do niej obiekt:

```

CREATE TABLE object_business_customers OF t_business_person;

INSERT INTO object_business_customers VALUES (
  t_business_person(1, 'Jan', 'Brązowy', 'Kierownik', 'XYZ SA')
);

```

Poniższe zapytanie wywołuje funkcję `display_details()` za pomocą `object_business_customers`:

```

SELECT o.display_details()
FROM object_business_customers o;

O.DISPLAY_DETAILS()
-----
id=1, name=Jan Brązowy, title=Kierownik, company=XYZ SA

```

Jak widać, wyświetlane są wartości `id`, `name` i `dob` (zwarcane przez funkcję `display_details()`, zdefiniowaną w `t_person`) oraz `title` i `company` (zwarcane przez funkcję `display_details()`, zdefiniowaną w `t_business_person`).

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- typ obiektowy może zawierać atrybuty i metody,
- za pomocą typu obiektowego możemy zdefiniować obiekt kolumnowy lub tabelę obiektową,
- tworzenie obiektów i praca z nimi mogą być wykonywane zarówno w języku SQL, jak i PL/SQL,
- w celu uzyskania dostępu do wiersza w tabeli obiektowej można użyć odwołania obiektowego,
- typy obiektowe mogą dziedziczyć od siebie atrybuty i metody,
- typ obiektowy można oznaczyć jako `NOT INSTANTIABLE`, co zapobiega tworzeniu obiektów tego typu,
- w konstruktorze można ustawić domyślne wartości atrybutów,

386 Oracle Database 12c i SQL. Programowanie

- można przesłonić metodę zdefiniowaną w typie nadrzędnym metodą zdefiniowaną w podtypie,
- mechanizm uogólnionego wywołania pozwala wywoływać z poziomu podtypu metody z typu nadrzędnego.

W kolejnym rozdziale zostaną opisane kolekcje.

ROZDZIAŁ

14

Kolekcje

W tym rozdziale:

- użyjemy kolekcji do definiowania kolumn w tabelach,
- będziemy tworzyć dane kolekcji w SQL oraz PL/SQL i pracować z nimi,
- dowiesz się, w jaki sposób kolekcja może zawierać osadzone kolekcje.

Podstawowe informacje o kolekcjach

Kolekcje umożliwiają składowanie zbiorów elementów w bazie danych. Są trzy typy kolekcji:

- **Typ VARRAY**, który jest podobny do tablicy w językach Java, C++ i C#. Przechowuje on zbiór elementów, a każdy z nich ma indeks określający jego pozycję w tablicy. Elementy z takiej tablicy mogą być modyfikowane jedynie jako całość, a nie pojedynczo. To oznacza, że jeżeli chcemy zmodyfikować tylko jeden element, i tak musimy przesłać wszystkie elementy tablicy. Typ VARRAY ma maksymalny rozmiar ustawiany przy jego tworzeniu, można go jednak potem zmienić.
- **Tabele zagnieżdżone**, które są tabelami zagnieżdżonymi w innych tabelach. Do tabeli zagnieżdżonej można wstawić pojedyncze elementy, modyfikować je i usuwać. To sprawia, że są one znacznie bardziej elastyczne niż typ VARRAY, w którym można modyfikować jedynie cały zbiór. Tabela zagnieżdżona nie ma maksymalnego rozmiaru, można więc w niej składować dowolną liczbę elementów.
- **Tablice asocjacyjne** (wcześniej nazywane tablicami indeksowanymi) przypominają tablice hashujące z języka Java. Tablica asocjacyjna, wprowadzona w Oracle Database 10g, stanowi zbiór par klucz-wartość. Wartość z tablicy możemy pobrać za pomocą klucza (który może być napisem) lub liczby całkowitej określającej pozycję wartości w tablicy. Tablica asocjacyjna może być używana jedynie w PL/SQL i nie może być składowana w bazie danych.

W tym rozdziale dokładniej omówimy tego typu kolekcje.

Uruchomienie skryptu tworzącego schemat bazy danych collection_schema

W katalogu SQL znajduje się skrypt *collection_schema.sql*. Tworzy on konto użytkownika o nazwie *collection_user* i hasło *collection_password* oraz kolekcje, tabele i kod PL/SQL, wykorzystywane w pierwnej części tego rozdziału.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika **system**.
3. Uruchomić skrypt *collection_schema.sql* w SQL*Plus za pomocą polecenia **@**.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu C:\SQL, to należy wpisać polecenie:

```
@ C:\SQL\collection_schema.sql
```

Po zakończeniu pracy skryptu będzie zalogowany użytkownik **collection_user**.

Tworzenie kolekcji

Z tego podrozdziału dowiesz się, jak tworzyć kolekcje i tabele zagnieżdżone.

Tworzenie typu VARRAY

W tablicy VARRAY jest składowany uporządkowany zbiór elementów tego samego typu (może to być typ wbudowany do bazy danych lub zdefiniowany przez użytkownika). Każdy element ma indeks, który określa jego pozycję w tablicy. Można modyfikować jedynie całą tablicę, a nie jej poszczególne elementy.

Do tworzenia typu VARRAY służy instrukcja **CREATE TYPE**, w której należy określić maksymalny rozmiar oraz typ elementów składowanych w tablicy. Poniższy przykład tworzy typ o nazwie **t_varray_address**, w którym mogą być składowane trzy napisy typu **VARCHAR2**:

```
CREATE TYPE t_varray_address AS VARRAY(3) OF VARCHAR2(50);
/
```

Każdy element **VARCHAR2** będzie reprezentował inny adres klienta naszego przykładowego sklepu.

W Oracle Database 10g i późniejszych można za pomocą instrukcji **ALTER TYPE** zmienić maksymalną liczbę elementów składowanych w tablicy **VARRAY**. Na przykład poniższa instrukcja zmienia maksymalną liczbę elementów na 10:

```
ALTER TYPE t_varray_address MODIFY LIMIT 10 CASCADE;
```

Dzięki opcji **CASCADE** zmiana propaguje do wszystkich obiektów zależnych w bazie danych.

Tworzenie tabeli zagnieżdżonej

W tabeli zagnieżdżonej jest składowany nieuporządkowany zbiór dowolnej liczby elementów. Można do niej wstawać pojedyncze elementy oraz aktualizować je i usuwać. Tabela zagnieżdżona nie posiada maksymalnego rozmiaru i można w niej składować dowolną liczbę elementów.

W tym podrozdziale utworzymy tabelę zagnieżdzoną, w której będą składowane obiekty typu **t_address**. Typ został opisany w poprzednim rozdziale — reprezentuje on adres i jest definiowany w następujący sposób:

```
CREATE TYPE t_address AS OBJECT (
  street VARCHAR2(15),
  city VARCHAR2(15),
  state CHAR(2),
  zip VARCHAR2(5)
);
/
```

Tabelę zagnieżdżoną tworzymy za pomocą instrukcji **CREATE TYPE**. Poniższa instrukcja tworzy typ o nazwie **t_nested_table_address**, w którym składowane są obiekty **t_address**:

```
CREATE TYPE t_nested_table_address AS TABLE OF t_address;
/
```

Należy zauważyć, że nie określono maksymalnego rozmiaru tabeli zagnieżdżonej — może ona przechowywać dowolną liczbę elementów.

Użycie kolekcji do definiowania kolumny w tabeli

Po utworzeniu typu kolekcji możemy za jego pomocą zdefiniować kolumnę w tabeli. Zobaczysz, jak użyć typu VARRAY i tabeli zagnieżdzonej, utworzonych w poprzednim podrozdziale, do zdefiniowania kolumny w tabeli.

Użycie typu VARRAY do zdefiniowania kolumny w tabeli

Poniższa instrukcja tworzy tabelę `customers_with_varray`, w której kolumna `addresses` jest definiowana przez typ `t_varray_address`:

```
CREATE TABLE customers_with_varray (
    id INTEGER PRIMARY KEY,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    addresses t_varray_address
);
```

Elementy z tablicy VARRAY są składowane bezpośrednio w tabeli, jeżeli rozmiar VARRAY nie przekracza 4 kilobajtów. W przeciwnym razie tablica VARRAY jest składowana poza tabelą. Jeżeli tablica VARRAY jest składowana w tabeli, uzyskanie dostępu do jej elementów jest szybsze niż sięganie po elementy z tabeli zagnieżdzonej.

Użycie typu tabeli zagnieżdzonej do zdefiniowania kolumny w tabeli

Poniższa instrukcja tworzy tabelę `customers_with_nested_table`, w której kolumna `addresses` jest definiowana przez typ `t_nested_table_address`:

```
CREATE TABLE customers_with_nested_table (
    id INTEGER PRIMARY KEY,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    addresses t_nested_table_address
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses;
```

Klauzula NESTED TABLE określa nazwę kolumny będącej tabelą zagnieżdzoną (w naszym przykładzie `addresses`), klauzula STORE AS określa natomiast nazwę tabeli zagnieżdzonej (w naszym przykładzie `nested_table`), w której są składowane faktyczne elementy. Do tabeli zagnieżdzonej nie można uzyskać dostępu niezależnie od tabeli, w której jest ona osadzona.

Uzyskiwanie informacji o kolekcjach

Jak się przekonamy w tym podrozdziale, informacje o kolekcjach możemy uzyskać za pomocą polecenia DESCRIBE, a także z kilku perspektyw użytkownika.

Uzyskiwanie informacji o tablicy VARRAY

Poniższy przykład prezentuje opis `t_varray_address`:

```
DESCRIBE t_varray_address
t_varray_address VARRAY(3) OF VARCHAR2(50)
```

Kolejny przykład prezentuje opis tabeli `customers_with_varray`, której kolumna `addresses` jest typu `t_varray_address`:

```
DESCRIBE customers_with_varray
```

Name	NULL?	Type
ID	NOT NULL	NUMBER(38)
FIRST_NAME		VARCHAR2(10)
LAST_NAME		VARCHAR2(10)
ADRESSES		T_VARRAY_ADDRESS

Informacje o obiektach VARRAY można również uzyskać z perspektywy `user_varrays`. Tabela 14.1 zawiera opis kolumn w tej perspektywie.

Tabela 14.1. Kolumny w perspektywie `user_varrays`

Kolumna	Typ	Opis
parent_table_name	VARCHAR2(128)	Nazwa tabeli zawierającej tablicę VARRAY
parent_table_column	VARCHAR2(4000)	Nazwa kolumny zawierającej tablicę VARRAY w tabeli nadzędnej
type_owner	VARCHAR2(128)	Nazwa użytkownika będącego właścicielem typu VARRAY
type_name	VARCHAR2(128)	Nazwa typu VARRAY
lob_name	VARCHAR2(128)	Nazwa dużego obiektu (LOB), jeżeli tablica VARRAY jest składowana w LOB. Duże obiekty zostaną opisane w następnym rozdziale
storage_spec	VARCHAR2(30)	Specyfikacja składowania tablicy VARRAY
return_type	VARCHAR2(20)	Typ zwracany przez kolumny
element_substitutable	VARCHAR2(25)	Okręsza, czy element VARRAY może być zastąpiony podtypem (Y lub N)

Poniższy przykład pobiera z perspektywy `user_varrays` informacje o `t_varray_address`:

```
SELECT parent_table_name, parent_table_column, type_name
FROM user_varrays
WHERE type_name = 'T_VARRAY_ADDRESS';
```

```
PARENT_TABLE_NAME
-----
PARENT_TABLE_COLUMN
-----
TYPE_NAME
-----
CUSTOMERS_WITH_VARRAY
ADDRESSES
T_VARRAY_ADDRESS
```

 **Uwaga** Informacje na temat wszystkich dostępnych elementów VARRAY można uzyskać z perspektywy `all_varrays`.

Uzyskiwanie informacji o tabeli zagnieżdzonej

Opis tabeli zagnieżdzonej można wyświetlić za pomocą polecenia `DESCRIBE`, co obrazuje poniższy przykład opisujący `t_nested_table_address`:

```
DESCRIBE t_nested_table_address
```

Name	NULL?	Type
STREET		VARCHAR2(15)
CITY		VARCHAR2(15)
STATE		CHAR(2)
ZIP		VARCHAR2(5)

Kolejny przykład zawiera opis tabeli `customers_with_nested_table`, w której kolumna `addresses` jest typu `t_nested_table_address`:

```
DESCRIBE customers_with_nested_table
```

Name	NULL?	Type
ID	NOT NULL	NUMBER(38)
FIRST_NAME		VARCHAR2(10)
LAST_NAME		VARCHAR2(10)
ADDRESSES		T_NESTED_TABLE_ADDRESS

Jeżeli ustawimy głębokość opisu na 2 i wyświetlimy opis tabeli `customers_with_nested_table`, zostaną również wyświetcone atrybuty typu `t_nested_table_address`:

```
SET DESCRIBE DEPTH 2
```

```
DESCRIBE customers_with_nested_table
```

Name	NULL?	Type
ID	NOT NULL	NUMBER(38)
FIRST_NAME		VARCHAR2(10)
LAST_NAME		VARCHAR2(10)
ADDRESSES		T_NESTED_TABLE_ADDRESS
STREET		VARCHAR2(15)
CITY		VARCHAR2(15)
STATE		CHAR(2)
ZIP		VARCHAR2(5)

Informacje o tabelach zagnieżdżonych można również pobrać z perspektywy `user_nested_tables`. Tabela 14.2 zawiera opis kolumn tej perspektywy.

Tabela 14.2. Kolumny w perspektywie `user_nested_tables`

Kolumna	Typ	Opis
table_name	VARCHAR2(128)	Nazwa tabeli zagnieżdżonej
table_type_owner	VARCHAR2(128)	Nazwa użytkownika będącego właścicielem typu tabeli zagnieżdżonej
table_type_name	VARCHAR2(128)	Nazwa typu tabeli zagnieżdżonej
parent_table_name	VARCHAR(128)	Nazwa tabeli nadzędnej, zawierającej tabelę zagnieżdżoną
parent_table_column	VARCHAR(4000)	Nazwa kolumny zawierającej tabelę zagnieżdżoną w tabeli nadzędnej
storage_spec	VARCHAR2(30)	Specyfikacja składowania VARRAY
return_type	VARCHAR2(20)	Typ zwracany przez kolumny
element_substitutable	VARCHAR2(25)	Określa, czy element VARRAY może być zastąpiony podtypem (Y lub N)

W poniższym przykładzie pobrano informacje o tabeli `nested_addresses` z perspektywy `user_nested_tables`:

```
SELECT table_name, table_type_name, parent_table_name, parent_table_column
FROM user_nested_tables
WHERE table_name = 'NESTED_ADDRESSES';
```

TABLE_NAME	TABLE_TYPE_NAME
PARENT_TABLE_NAME	
PARENT_TABLE_COLUMN	
NESTED_ADDRESSES	T_NESTED_TABLE_ADDRESS
CUSTOMERS_WITH_NESTED_TABLE	
ADDRESSES	



Informacje o wszystkich dostępnych tabelach zagnieżdżonych można pobrać z perspektywy `all_nested_tables`.

Umieszczanie elementów w kolekcji

Z tego podrozdziału dowiesz się, jak za pomocą instrukcji INSERT umieszcza się elementy w tablicy VARRAY i tabeli zagnieżdzonej. Nie jest konieczne uruchamianie instrukcji INSERT prezentowanych w tym podroziale, ponieważ zostały one uruchomione przez skrypt *collection_schema.sql*.

Umieszczanie elementów w tablicy VARRAY

Poniższe instrukcje INSERT wstawiają wiersze do tabeli *customers_with_varray*. Należy zauważyć użycie konstruktora *t_varray_address* w celu określenia napisów umieszczanych jako elementy tablicy VARRAY:

```
INSERT INTO customers_with_varray VALUES (
  1, 'Stefan', 'Brązowy',
  t_varray_address(
    'Stanowa 2, Fasolowo, MAZ, 12345',
    'Górskiego 4, Rzeszotary, GDA, 54321'
  )
);

INSERT INTO customers_with_varray VALUES (
  2, 'Jan', 'Kowalski',
  t_varray_address(
    'Wysoka 1, Załęże, MAŁ, 12347',
    'Nowa 4, Byczyna, MAZ, 54323',
    'Handlowa 7, Zarzecze, POZ, 54323'
  )
);
```

W pierwszym wierszu znajdują się dwa adresy, a w drugim wierszu są trzy. Można składować dowolną liczbę adresów nieprzekraczającą maksymalnego rozmiaru tablicy VARRAY.

Umieszczanie elementów w tabeli zagnieżdzonej

Poniższe instrukcje INSERT wstawiają wiersze do tabeli *customers_with_nested_table*. Należy zauważyć użycie konstruktorów *t_nested_table_address* i *t_address* do określenia elementów tabeli zagnieżdzonej:

```
INSERT INTO customers_with_nested_table VALUES (
  1, 'Stefan', 'Brązowy',
  t_nested_table_address(
    t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
    t_address('Górskiego 4', 'Rzeszotary', 'GDA', '54321')
  )
);

INSERT INTO customers_with_nested_table VALUES (
  2, 'Jan', 'Kowalski',
  t_nested_table_address(
    t_address('Wysoka 1', 'Załęże', 'MAŁ', '12347'),
    t_address('Nowa 4', 'Byczyna', 'MAZ', '54323'),
    t_address('Handlowa 7', 'Zarzecze', 'POZ', '54323')
  )
);
```

Pierwszy wiersz zawiera dwa adresy, a drugi trzy. W tabeli zagnieżdzonej można składować dowolną liczbę adresów.

Pobieranie elementów z kolekcji

W tym podroziale nauczysz się, jak pobierać elementy z tablic VARRAY i tabel zagnieżdzonych za pomocą zapytań. Wyniki zwracane przez zapytania zostały nieco sformatowane w celu zwiększenia czytelności.

Pobieranie elementów z tablicy VARRAY

Poniższe zapytanie pobiera informacje o kliencie nr 1 z tabeli `customers_with_varray`. Zwracany jest jeden wiersz zawierający oba adresy składowane w tablicy VARRAY:

```
SELECT *
FROM customers_with_varray
WHERE id = 1;

ID FIRST_NAME LAST_NAME
-----
ADDRESSES
-----
1 Stefan Brązowy
T_VARRAY_ADDRESS('Stanowa 2, Fasolowo, MAZ, 12345',
'Górska 4, Rzeszotary, GDA, 54321')
```

W kolejnym zapytaniu wyszczególniono nazwy kolumn:

```
SELECT id, first_name, last_name, addresses
FROM customers_with_varray
WHERE id = 1;
```

```
ID FIRST_NAME LAST_NAME
-----
ADDRESSES
-----
1 Stefan Brązowy
T_VARRAY_ADDRESS('Stanowa 2, Fasolowo, MAZ, 12345',
'Górska 4, Rzeszotary, GDA, 54321')
```

W powyższych przykładach adresy składowane w tablicy VARRAY są zwracane jako jeden wiersz. Z podrozdziału „Użycie funkcji TABLE() do interpretacji kolekcji jako serii wierszy” dowiesz się, w jaki sposób można zinterpretować dane składowane w kolekcji jako serię wierszy.

Pobieranie elementów z tabeli zagnieżdżonej

Poniższe zapytanie pobiera informacje o kliencie nr 1 z tabeli `customers_with_nested_table`. Zwracany jest jeden wiersz, który zawiera dwa adresy składowane w tabeli zagnieżdżonej:

```
SELECT *
FROM customers_with_nested_table
WHERE id = 1;

ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP)
-----
1 Stefan Brązowy
T_NESTED_TABLE_ADDRESS(
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))
```

W kolejnym zapytaniu wyszczególniono nazwy kolumn:

```
SELECT id, first_name, last_name, addresses
FROM customers_with_nested_table
WHERE id = 1;
```

```
ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP)
-----
1 Stefan Brązowy
```

```
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
  T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))
```

Poniższe zapytanie pobiera jedynie tabele zagnieżdżoną `address`. Podobnie jak w poprzednich przykładach zwracany jest jeden wiersz zawierający dwa adresy składowane w tabeli zagnieżdżonej:

```
SELECT addresses
FROM customers_with_nested_table
WHERE id = 1;

ADDRESSES(STREET, CITY, STATE, ZIP)
-----
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
  T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))
```

Użycie funkcji TABLE() do interpretacji kolekcji jako serii wierszy

Zapytania prezentowane dotychczas w tym rozdziale zwracały zawartość kolekcji jako jeden wiersz. Czasami chcemy jednak potraktować dane składowane w kolekcji jako serię wierszy. Możemy na przykład pracować ze starszą aplikacją wykorzystującą jedynie wiersze. Funkcja `TABLE()` pozwala zinterpretować kolekcję jako serię wierszy. Z tego podrozdziału dowiesz się, jak stosować tę funkcję dla typów `VARRAY` i tabeli zagnieżdżonych.

Użycie funkcji TABLE() z typem VARRAY

W poniższym zapytaniu użyto funkcji `TABLE()` do pobrania z tabeli `customers_with_varray` dwóch adresów klienta nr 1. Zostały zwrócone dwa odrębne wiersze:

```
SELECT a.*
FROM customers_with_varray c, TABLE(c.addresses) a
WHERE id = 1;

COLUMN_VALUE
-----
Stanowa 2, Fasolowo, MAZ, 12345
Górska 4, Rzeszotary, GDA, 54321
```

Należy zauważać, że oprogramowanie Oracle automatycznie dodaje nazwę kolumny `COLUMN_VALUE` do wierszy zwróconych przez zapytanie.

`COLUMN_VALUE` jest aliasem pseudokolumny, który jest automatycznie dodawany, jeżeli kolekcja zawiera dane mające jeden z typów wbudowanych, na przykład `VARCHAR2`, `CHAR`, `NUMBER` lub `DATE`. Ponieważ w przykładzie tablica `VARRAY` zawiera dane typu `VARCHAR2`, został dodzony alias `COLUMN_VALUE`. Gdyby tablica zawierała dane o typie zdefiniowanym przez użytkownika, funkcja `TABLE()` zwróciłaby obiekty tego typu bez dołączonego aliasu `COLUMN_VALUE`, o czym przekonasz się w następnym podrozdziale.

W funkcji `TABLE()` można również osadzić całą instrukcję `SELECT`. Na przykład poniższe zapytanie zmienia wcześniejsze przez umieszczenie instrukcji `SELECT` wewnątrz funkcji `TABLE()`:

```
SELECT *
FROM TABLE(
  -- pobiera adresy klienta nr 1
  SELECT addresses
  FROM customers_with_varray
  WHERE id = 1
);

COLUMN_VALUE
```

```
-----  
Stanowa 2, Fasolowo, MAZ, 12345  
Górsko 4, Rzeszotary, GDA, 54321
```

Następne zapytanie stanowi kolejny przykład użycia funkcji TABLE() do pobrania adresów:

```
SELECT c.id, c.first_name, c.last_name, a.*  
FROM customers_with_varray c, TABLE(c.addresses) a  
WHERE id = 1;
```

```
ID FIRST_NAME LAST_NAME  
-----  
COLUMN_VALUE  
-----  
1 Stefan Brązowy  
Stanowa 2, Fasolowo, MAZ, 12345  
  
1 Stefan Brązowy  
Górsko 4, Rzeszotary, GDA, 54321
```

Użycie funkcji TABLE() z tabelą zagnieżdzoną

W poniższym zapytaniu użyto funkcji TABLE() do pobrania z tabeli `customers_with_nested_table` dwóch adresów klienta nr 1:

```
SELECT a.*  
FROM customers_with_nested_table c, TABLE(c.addresses) a  
WHERE id = 1;
```

```
STREET CITY STA ZIP  
-----  
Stanowa 2 Fasolowo MAZ 12345  
Górsko 4 Rzeszotary GDA 54321
```

Kolejne zapytanie pobiera atrybuty street i state adresów:

```
SELECT a.street, a.state  
FROM customers_with_nested_table c, TABLE(c.addresses) a  
WHERE id = 1;
```

```
STREET STA  
-----  
Stanowa 2 MAZ  
Górsko 4 GDA
```

Poniższe zapytanie stanowi kolejny przykład użycia funkcji TABLE() do pobrania adresów:

```
SELECT c.id, c.first_name, c.last_name, a.*  
FROM customers_with_nested_table c, TABLE(c.addresses) a  
WHERE c.id = 1;
```

```
ID FIRST_NAME LAST_NAME STREET CITY STA ZIP  
-----  
1 Stefan Brązowy Stanowa 2 Fasolowo MAZ 12345  
1 Stefan Brązowy Górska 4 Rzeszotary GDA 54321
```

W podrozdziale „Modyfikowanie elementów tabeli zagnieżdzionej” zostanie opisane bardzo istotne zastosowanie funkcji TABLE().

Modyfikowanie elementów kolekcji

W tym podrozdziale opisano, w jaki sposób modyfikuje się elementy w tablicy VARRAY i tabeli zagnieżdzonej. Instrukcje UPDATE, INSERT i DELETE, prezentowane w tym podrozdziale, można oczywiście uruchamiać samodzielnie.

Modyfikowanie elementów tablicy VARRAY

Elementy składowane w tablicy VARRAY mogą być modyfikowane jedynie jako całość, co oznacza, że nawet jeżeli chcemy zmodyfikować tylko jeden element, musimy przesyłać wszystkie elementy tej struktury. Poniższa instrukcja UPDATE modyfikuje adresy klienta nr 2 w tabeli `customers_with_varray`:

```
UPDATE customers_with_varray
SET addresses = t_varray address(
  'Dowolna 6, Lednica, POM, 33347',
  'Nowa 4, Byczyna, MAZ, 54323',
  'Handlowa 7, Zarzecze, POZ, 54323'
)
WHERE id = 2;
```

1 row updated.

Modyfikowanie elementów tabeli zagnieżdzonej

W przeciwieństwie do tablic VARRAY elementy tabeli zagnieżdzonej mogą być modyfikowane pojedynczo. Do tabeli zagnieżdzonej można wstawać poszczególne elementy, modyfikować je i usuwać. W tym podrozdziale zostaną opisane te trzy operacje.

Poniższa instrukcja INSERT wstawia do tabeli `customers_with_nested_table` kolejny adres klienta nr 2. Należy zauważyć, że do pobrania adresów jako serii wierszy użyto funkcji TABLE():

```
INSERT INTO TABLE(
  -- pobiera adresy klienta nr 2
  SELECT addresses
  FROM customers_with_nested_table
  WHERE id = 2
) VALUES (
  t_address('Główna 5', 'Kaminy', 'MAZ', '55512')
);
```

1 row created.

Poniższa instrukcja UPDATE zmienia w adresie klienta nr 2 adres `Wysoka 1` na `Dowolna 9`. Należy zauważyc użycie aliasu `addr` w klauzulach VALUE przy określaniu wartości:

```
UPDATE TABLE(
  -- pobiera adresy klienta nr 2
  SELECT addresses
  FROM customers_with_nested_table
  WHERE id = 2
) addr
SET VALUE(addr) =
  t_address('Dowolna 9', 'Wolnowo', 'POM', '74321')
WHERE VALUE(addr) =
  t_address('Wysoka 1', 'Załęże', 'MAŁ', '12347');
```

1 row updated.

Poniższa instrukcja DELETE usuwa adres `Nowa 4...` klienta nr 2:

```
DELETE FROM TABLE(
  -- pobiera adresy klienta nr 2
  SELECT addresses
  FROM customers_with_nested_table
  WHERE id = 2
) addr
WHERE VALUE(addr) =
  t_address('Nowa 4', 'Byczyna', 'MAZ', '54323');
```

1 row deleted.

Użycie metody mapującej do porównywania zawartości tabel zagnieżdzonych

Możliwe jest porównywanie zawartości tabeli zagnieżdzonej z zawartością innej tabeli zagnieżdzonej. Dwie tabele zagnieżdzone są równe, jeżeli łącznie spełnione są poniższe warunki:

- tabele są tego samego typu,
- mają tę samą liczbę wierszy,
- wszystkie ich elementy mają te same wartości.

Jeżeli elementy w tabeli zagnieżdzonej są o typie wbudowanym do bazy danych (na przykład NUMBER, VARCHAR2 itd.), baza danych potrafi automatycznie porównać zawartość tabel zagnieżdzonych. Jeżeli jednak elementy są o typie zdefiniowanym przez użytkownika, konieczne jest zapewnienie funkcji mapującej w celu porównania obiektów (funkcje mapujące zostały przedstawione w podrozdziale „Porównywanie wartości obiektów” poprzedniego rozdziału).

Poniższe instrukcje tworzą typ o nazwie `t_address2`, zawierający funkcję mapującą o nazwie `get_string()`. Należy zauważyć, że funkcja `get_string()` zwraca typ VARCHAR2, zawierający wartości atrybutów `zip`, `state`, `city` i `street`:

```
CREATE TYPE t_address2 AS OBJECT (
    street VARCHAR2(15),
    city VARCHAR2(15),
    state CHAR(3),
    zip VARCHAR2(5),

    -- deklaracja funkcji mapującej get_string()
    -- zwracającej napis VARCHAR2
    MAP MEMBER FUNCTION get_string RETURN VARCHAR2
);
/

CREATE TYPE BODY t_address2 AS
    -- definicja funkcji mapującej get_string()
    MAP MEMBER FUNCTION get_string RETURN VARCHAR2 IS
        BEGIN
            -- zwraca skonkatenowany napis, zawierający wartości
            -- atrybutów zip, state, city i street
            RETURN zip || ' ' || state || ' ' || city || ' ' || street;
        END get_string;
    END;
/

```

Jak się wkrótce przekonasz, baza danych automatycznie wywoła funkcję `get_string()` przy porównywaniu obiektów typu `t_address2`.

Poniższe instrukcje tworzą typ tabeli zagnieżdzonej i wstawiają do niej wiersz:

```
CREATE TYPE t_nested_table_address2 AS TABLE OF t_address2;
/

CREATE TABLE customers_with_nested_table2 (
    id INTEGER PRIMARY KEY,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    addresses t_nested_table_address2
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses2;
```

```
INSERT INTO customers_with_nested_table2 VALUES (
  1, 'Stefan', 'Brązowy',
  t_nested_table_address2(
    t_address2('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
    t_address2('Wysoka 4', 'Byczyna', 'POZ', '54321')
  )
);
```

W klauzuli WHERE poniższego zapytania została umieszczona tabela zagnieżdzona. Należy zauważać, że adresy określone za operatorem = w klauzuli WHERE są takie same jak w powyższej instrukcji INSERT:

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
WHERE cn.addresses =
  t_nested_table_address2(
    t_address2('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
    t_address2('Wysoka 4', 'Byczyna', 'POZ', '54321')
);
```

ID	FIRST_NAME	LAST_NAME
1	Stefan	Brązowy

Po uruchomieniu tego zapytania baza danych automatycznie wywołuje funkcję `get_string()` w celu porównania obiektów typu `t_address2` z `cn.addresses` z obiektami typu `t_address2` za operatorem = w klauzuli WHERE. Funkcja `get_string()` zwraca napis `VARCHAR2`, zawierający wartości atrybutów `zip`, `state`, `city` i `street` obiektów, a jeżeli dla każdego obiektu napisy są równe, tabele zagnieżdzione również są równe.

Kolejne zapytanie nie zwraca żadnych wierszy, ponieważ jednemu adresowi umieszczonemu za operatorem = w klauzuli WHERE odpowiada tylko jeden adres w `cn.addresses` (jak pamiętamy, dwie tabele zagnieżdzone są równe wtedy i tylko wtedy, gdy są tego samego typu, mają tę samą liczbę wierszy, a ich elementy mają te same wartości):

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
WHERE cn.addresses =
  t_nested_table_address2(
    t_address2('Wysoka 4', 'Byczyna', 'POZ', '54321')
);
```

no rows selected

W Oracle Database 10g i późniejszych można użyć operatora SUBMULTISET do sprawdzenia, czy wartość jednej tabeli zagnieżdzonej jest podzbiorem innej tabeli zagnieżdzonej. Poniżej zmieniono przedstawione zapytanie i zwracany jest jeden wiersz:

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
WHERE
  t_nested_table_address2(
    t_address2('Wysoka 4', 'Byczyna', 'POZ', '54321')
  )
SUBMULTISET OF cn.addresses;
```

ID	FIRST_NAME	LAST_NAME
1	Stefan	Brązowy

Ponieważ adres w pierwszej części klauzuli WHERE jest podzbiorem adresów w `cn.addresses`, zwracany jest jeden wiersz.

Poniższe zapytanie stanowi kolejny przykład. Tym razem adresy w `cn.addresses` są podzbiorem adresów wymienionych po słowie kluczowym OF w klauzuli WHERE:

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
```

```

WHERE
  cn.addresses SUBMULTISET OF
  t_nested_table_address2(
    t_address2('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
    t_address2('Wysoka 4', 'Byczyna', 'POZ', '54321'),
    t_address2('Stanowa 6', 'Fasolowo', 'MAZ', '12345')
  );

```

ID	FIRST_NAME	LAST_NAME
1	Stefan	Brązowy

Operator **SUBMULTISET** zostanie opisany szczegółowo w podrozdziale „Operator SUBMULTISET”, w dalszej części tego rozdziału. Z podrozdziału „Operatory równości i nierówności” dowiesz się, jak stosować operatory ANSI zaimplementowane w Oracle Database 10g do porównywania tabel zagnieżdżonych.

 **Nie istnieje mechanizm bezpośredniego porównywania zawartości tablic VARRAY.**

Uwaga

Użycie funkcji **CAST** do konwersji kolekcji z jednego typu na inny

Z pomocą funkcji **CAST()** możemy przekonwertować kolekcję jednego typu na inny. Z tego podrozdziału dowiesz się, w jaki sposób można za pomocą tej funkcji przekonwertować tablicę **VARRAY** na tabelę zagnieżdżoną i odwrotnie.

Użycie funkcji **CAST()** do konwersji tablicy **VARRAY** na tabelę zagnieżdżoną

Poniższe instrukcje tworzą i umieszczają dane w tabeli **customers_with_varray2**, zawierającej kolumnę **addresses** o typie **t_varray_address2**:

```

CREATE TYPE t_varray_address2 AS VARRAY(3) OF t_address;
/
CREATE TABLE customers_with_varray2 (
  id INTEGER PRIMARY KEY,
  first_name VARCHAR2(10),
  last_name VARCHAR2(10),
  addresses t_varray_address2
);
INSERT INTO customers_with_varray2 VALUES (
  1, 'Jan', 'Bond',
  t_varray_address2(
    t_address('Newtona 3', 'Inny Kącik', 'MAŁ', '22123'),
    t_address('Wiosenna 6', 'Nowe Miasto', 'POZ', '77712')
  )
);

```

W poniższym zapytaniu użyto funkcji **CAST()** w celu zwrócenia adresów klienta nr 1, składowanych w tablicy **VARRAY** jako tabeli zagnieżdżonej. Należy zauważyć, że adresy znajdują się w konstruktorze typu **T_NESTED_TABLE_ADDRESS**, co wskazuje na konwersję elementów na ten typ:

```

SELECT CAST(cv.addresses AS t_nested_table_address)
FROM customers_with_varray2 cv
WHERE cv.id = 1;

```

```
CAST(CV.ADDRESSESAS T_NESTED_TABLE_ADDRESS)(STREET, CITY, STATE, ZIP)
```

```
-- T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Newtona 3', 'Inny Kącik', 'MAŁ', '22123'),
  T_ADDRESS('Wiosenna 6', 'Nowe Miasto', 'POZ', '77712'))
```

Użycie funkcji CAST() do konwersji tabeli zagnieżdzonej na tablicę VARRAY

W poniższym zapytaniu użyto funkcji CAST() w celu zwrócenia adresów klienta nr 1 z tabeli `customers_with_nested_table` jako tablicy VARRAY. Należy zauważyć, że adresy są umieszczone w konstruktorze `T_VARRAY_ADDRESS2`:

```
SELECT CAST(cn.addresses AS t_varray_address2)
FROM customers_with_nested_table cn
WHERE cn.id = 1;
```

```
CAST(cn.addresses AS T_VARRAY_ADDRESS2) (STREET, CITY, STATE, ZIP)
```

```
T_VARRAY_ADDRESS2(
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
  T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))
```

Użycie kolekcji w PL/SQL

Kolekcje mogą być wykorzystywane w języku PL/SQL. W tym podrozdziale nauczysz się, jak:

- manipulować tablicą VARRAY,
- manipulować tabelą zagnieżdzoną,
- uzyskiwać dostęp do kolekcji i manipulować nimi za pomocą metod kolekcji dostępnych w PL/SQL.

Wszystkie pakiety prezentowane w tym rozdziale zostały utworzone przez skrypt `collection_schema.sql`. Jeżeli wykonano instrukcje `INSERT`, `UPDATE` lub `DELETE`, przedstawione w poprzednim podrozdziale, należy ponownie uruchomić skrypt `collection_schema.sql`, aby uzyskiwane wyniki były zgodne z prezentowanymi w książce.

Manipulowanie tablicą VARRAY

W tym podrozdziale będziemy pracować z pakietem `varray_package`. Zawiera on następujące elementy:

- typ `REF CURSOR` o nazwie `t_ref_cursor`,
- funkcję `get_customers()`, zwracającą obiekt typu `t_ref_cursor` wskazujący na wiersze w tabeli `customers_with_varray`,
- procedurę `insert_customer()`, wstawiającą wiersz do tabeli `customers_with_varray`.

Skrypt `collection_schema.sql` zawiera następującą specyfikację i treść pakietu `varray_package`:

```
CREATE PACKAGE varray_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_customers RETURN t_ref_cursor;
  PROCEDURE insert_customer(
    p_id IN customers_with_varray.id%TYPE,
    p_first_name IN customers_with_varray.first_name%TYPE,
    p_last_name IN customers_with_varray.last_name%TYPE,
    p_addresses IN customers_with_varray.addresses%TYPE
  );
END varray_package;
/
CREATE PACKAGE BODY varray_package AS
```

```
-- funkcja get_customers() zwraca REF CURSOR
-- wskazujący na wiersze w customers_with_varray
FUNCTION get_customers
RETURN t_ref_cursor IS
    -- deklaracja obiektu REF CURSOR
    v_customers_ref_cursor t_ref_cursor;
BEGIN
    -- pobranie REF CURSOR
    OPEN v_customers_ref_cursor FOR
        SELECT *
        FROM customers_with_varray;
    -- zwraca REF CURSOR
    RETURN v_customers_ref_cursor;
END get_customers;

-- procedura insert_customer() wstawia wiersz
-- do tabeli customers_with_varray
PROCEDURE insert_customer(
    p_id IN customers_with_varray.id%TYPE,
    p_first_name IN customers_with_varray.first_name%TYPE,
    p_last_name IN customers_with_varray.last_name%TYPE,
    p_addresses IN customers_with_varray.addresses%TYPE
) IS
BEGIN
    INSERT INTO customers_with_varray
    VALUES (p_id, p_first_name, p_last_name, p_addresses);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END insert_customer;
END varray_package;
/

```

W poniższym przykładzie wywołano procedurę `insert_customer()` w celu wstawienia nowego wiersza do tabeli `customers_with_varray`:

```
CALL varray_package.insert_customer(
    3, 'Michał', 'Czerwony',
    t_varray_address(
        'Główna 10, Zielonki, MAŁ, 22212',
        'Stanowa 20, Zalewki, POZ, 22213'
    )
);
```

Call completed.

W kolejnym przykładzie wywołano funkcję `get_customers()` w celu pobrania wierszy z tabeli `customers_with_varray`:

```
SELECT varray_package.get_customers
FROM dual;
```

GET_CUSTOMERS

CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

ID FIRST_NAME LAST_NAME

ADDRESSES

1 Stefan Brązowy

```
T_VARRAY_ADDRESS('Stanowa 2, Fasolowo, MAZ, 12345',
  'Góriska 4, Rzeszotary, GDA, 54321')
```

```
2 Jan Kowalski
T_VARRAY_ADDRESS('Wysoka 1, Załęże, MAŁ, 12347',
  'Nowa 4, Byczyna, MAZ, 54323',
  'Handlowa 7, Zarzecze, POZ, 54323')
```

```
3 Michał Czerwony
T_VARRAY_ADDRESS('Główna 10, Zielonki, MAŁ, 22212',
  'Stanowa 20, Zalewki, POZ, 22213')
```

Manipulowanie tabelą zagnieżdzoną

W tym podrozdziale będziemy pracować z pakietem `nested_table_package`, który zawiera następujące elementy:

- typ `REF CURSOR` o nazwie `t_ref_cursor`,
- funkcję `get_customers()`, zwracającą obiekt typu `t_ref_cursor` wskazujący na wiersze w tabeli `customers_with_nested_table`,
- procedurę `insert_customer()`, wstawiającą wiersz do tabeli `customers_with_nested_table`.

Skrypt `collection_schema.sql` zawiera następującą specyfikację i treść pakietu `nested_table_package`:

```
CREATE PACKAGE nested_table_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_customers RETURN t_ref_cursor;
  PROCEDURE insert_customer(
    p_id IN customers_with_nested_table.id%TYPE,
    p_first_name IN customers_with_nested_table.first_name%TYPE,
    p_last_name IN customers_with_nested_table.last_name%TYPE,
    p_addresses IN customers_with_nested_table.addresses%TYPE
  );
END nested_table_package;
/

CREATE PACKAGE BODY nested_table_package AS
  -- funkcja get_customers() zwraca REF CURSOR
  -- wskazujący na wiersze tabeli customers_with_nested_table
  FUNCTION get_customers
  RETURN t_ref_cursor IS
    -- deklaracja obiektu REF CURSOR
    v_customers_ref_cursor t_ref_cursor;
  BEGIN
    -- pobranie REF CURSOR
    OPEN v_customers_ref_cursor FOR
      SELECT *
      FROM customers_with_nested_table;
    -- zwraca REF CURSOR
    RETURN v_customers_ref_cursor;
  END get_customers;

  -- procedura insert_customer() wstawia wiersz
  -- do tabeli customers_with_nested_table
  PROCEDURE insert_customer(
    p_id IN customers_with_nested_table.id%TYPE,
    p_first_name IN customers_with_nested_table.first_name%TYPE,
    p_last_name IN customers_with_nested_table.last_name%TYPE,
    p_addresses IN customers_with_nested_table.addresses%TYPE
  ) IS
  BEGIN
    INSERT INTO customers_with_nested_table
```

```

VALUES (p_id, p_first_name, p_last_name, p_addresses);
COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END insert_customer;
END nested_table_package;
/

```

W poniższym przykładzie wywołano procedurę `insert_customer()` w celu wstawienia nowego wiersza do tabeli `customers_with_nested_table`:

```

CALL nested_table_package.insert_customer(
  3, 'Michał', 'Czerwony',
  t_nested_table_address(
    t_address('Główna 10', 'Zielonki', 'MAŁ', '22212'),
    t_address('Stanowa 20', 'Zalewki', 'POZ', '22213')
  )
);

```

Call completed.

W kolejnym przykładzie wywołano funkcję `get_customers()` w celu pobrania wierszy z tabeli `customers_with_nested_table`:

```

SELECT nested_table_package.get_customers
FROM dual;

```

```

GET_CUSTOMERS
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP)
-----
1 Stefan Brązowy
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
  T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))

2 Jan Kowalski
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Wysoka 1', 'Załęże', 'MAŁ', '12347'),
  T_ADDRESS('Nowa 4', 'Byczyna', 'MAZ', '54323'),
  T_ADDRESS('Handlowa 7', 'Zarzecze', 'POZ', '54323'))

3 Michał Czerwony
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('Główna 10', 'Zielonki', 'MAŁ', '22212'),
  T_ADDRESS('Stanowa 20', 'Zalewki', 'POZ', '22213'))

```

Metody operujące na kolekcjach w PL/SQL

W tym podrozdziale poznasz metody PL/SQL operujące na kolekcjach. Tabela 14.3 zawiera opis tych metod. Mogą one być używane wyłącznie w języku PL/SQL.

W kolejnych podrozdziałach jest wykorzystywany pakiet o nazwie `collection_method_examples`. Przykłady ilustrują użycie metod opisanych w tabeli 14.3. Pakiet jest tworzony przez skrypt `collection_schema.sql`. Kod poszczególnych metod definiowanych w tym pakiecie będzie prezentowany w kolejnych podrozdziałach.

Tabela 14.3. Metody PL/SQL operujące na kolekcjach

Metoda	Opis
COUNT	Zwraca liczbę elementów w kolekcji. Ponieważ tabela zagnieżdzona może zawierać puste elementy, COUNT zwraca liczbę niepustych elementów tabeli zagnieżdzonych
DELETE DELETE(<i>n</i>) DELETE(<i>n</i> , <i>m</i>)	Usuwa elementy z kolekcji. Występują trzy odmiany metody DELETE: <ul style="list-style-type: none"> ■ DELETE usuwa wszystkie elementy ■ DELETE(<i>n</i>) usuwa element <i>n</i> ■ DELETE(<i>n</i>, <i>m</i>) usuwa elementy od <i>n</i> do <i>m</i> Ponieważ tablice VARRAY mają seryjne indeksy, nie można usuwać pojedynczych elementów z tablicy (oprócz elementów znajdujących się na końcu tablicy — służy do tego metoda TRIM)
EXISTS(<i>n</i>)	Zwraca true, jeżeli w kolekcji istnieje element <i>n</i> . Metoda EXISTS zwraca true dla niepustych elementów i false dla pustych elementów tabeli zagnieżdzonych lub znajdujących się poza zakresem kolekcji
EXTEND EXTEND(<i>n</i>) EXTEND(<i>n</i> , <i>m</i>)	Wstawia element na końcu kolekcji. Występują trzy odmiany metody EXTEND: <ul style="list-style-type: none"> ■ EXTEND wstawia jeden element o wartości NULL ■ EXTEND(<i>n</i>) wstawia <i>n</i> elementów o wartości NULL ■ EXTEND(<i>n</i>, <i>m</i>) wstawia <i>n</i> elementów o wartości skopiowanej z elementu <i>m</i>
FIRST	Zwraca indeks pierwszego elementu kolekcji. Jeżeli kolekcja jest pusta, metoda FIRST zwraca NULL. Ponieważ w tabeli zagnieżdzonej pojedyncze elementy mogą być puste, metoda FIRST zwraca najmniejszy indeks niepstego elementu tabeli zagnieżdzonej
LAST	Zwraca indeks ostatniego elementu kolekcji. Jeżeli kolekcja jest pusta, metoda FIRST zwraca NULL. Ponieważ w tabeli zagnieżdzonej pojedyncze elementy mogą być puste, metoda FIRST zwraca największy indeks niepstego elementu tabeli zagnieżdzonej
LIMIT	W przypadku tabel zagnieżdzonych o niezadeklarowanym rozmiarze metoda LIMIT zwraca NULL. W przypadku tablic VARRAY metoda LAST zwraca maksymalną liczbę elementów możliwą do przechowania w tablicy. Limit można określić w definicji typu. Jest on zmieniany przez metody TRIM i EXTEND, a także za pomocą instrukcji ALTER TYPE
NEXT(<i>n</i>)	Zwraca indeks elementu znajdującego się za elementem <i>n</i> . Ponieważ pojedyncze elementy tabeli zagnieżdzonej mogą być puste, metoda NEXT zwraca indeks niepstego elementu znajdującego się po elemencie <i>n</i> . Jeżeli po <i>n</i> nie ma żadnych elementów, metoda NEXT zwraca NULL
PRIOR(<i>n</i>)	Zwraca indeks elementu znajdującego się przed elementem <i>n</i> . Ponieważ pojedyncze elementy tabeli zagnieżdzonej mogą być puste, metoda PRIOR zwraca indeks niepstego elementu znajdującego się przed elementem <i>n</i> . Jeżeli przed <i>n</i> nie ma żadnych elementów, metoda PRIOR zwraca NULL
TRIM TRIM(<i>n</i>)	Usuwa elementy z końca kolekcji. Występują dwie odmiany metody TRIM: <ul style="list-style-type: none"> ■ TRIM usuwa jeden element z końca kolekcji ■ TRIM(<i>n</i>) usuwa <i>n</i> elementów z końca kolekcji

Metoda COUNT()

Metoda COUNT() zwraca liczbę elementów w kolekcji. Ponieważ w tabeli zagnieżdzonej mogą występować puste elementy, metoda COUNT() zwraca liczbę niepustych elementów w tabeli zagnieżdzonej. Założmy, że mamy tabelę zagnieżdzoną o nazwie *v_nested_table*, której elementy mają wartości przedstawione poniżej:

Indeks elementu	Pusty/niepusty
1	pusty
2	niepusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie `v_nested_table.COUNT` zwróci 2 — liczbę niepustych elementów.

Metoda `COUNT()` jest używana w metodach `get_addresses()` i `display_addresses()` pakietu `collection_method_examples`. Funkcja `get_addresses()` zwraca z tabeli `customers_with_nested_mtable` adres określonego klienta, którego `id` jest przesyłany do funkcji:

```
FUNCTION get_addresses(
    p_id customers_with_nested_table.id%TYPE
) RETURN t_nested_table_address IS
-- deklaracja obiektu o nazwie v_addresses,
-- w którym będzie składowana tabela zagnieżdzona z adresami
    v_addresses t_nested_table_address;
BEGIN
    -- pobranie tabeli zagnieżdzonej z adresami do v_addresses
    SELECT addresses
    INTO v_addresses
    FROM customers_with_nested_table
    WHERE id = p_id;

    -- wyświetla liczbę adresów za pomocą v_addresses.COUNT
    DBMS_OUTPUT.PUT_LINE(
        'Liczba adresów = ' || v_addresses.COUNT
    );

    -- zwraca v_addresses
    RETURN v_addresses;
END get_addresses;
```

Poniższy przykład wyłącza wypisywanie serwera oraz wywołuje funkcję `get_addresses()` dla klienta nr 1:

```
SET SERVEROUTPUT ON
SELECT collection_method_examples.get_addresses(1) addresses
FROM dual;
```

```
ADDRESSES(STREET, CITY, STATE, ZIP)
-----T_NESTED_TABLE_ADDRESS(
T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345'),
T_ADDRESS('Górska 4', 'Rzeszotary', 'GDA', '54321'))
```

Liczba adresów = 2

Poniższa procedura `display_addresses()` przyjmuje parametr o nazwie `p_addresses`, zawierający tabelę zagnieżdzoną z adresami. Procedura wyświetla liczbę adresów w tabeli `p_addresses`, używając metody `COUNT`, a następnie wyświetla te adresy, wykorzystując pętlę:

```
PROCEDURE display_addresses(
    p_addresses t_nested_table_address
) IS
    v_count INTEGER;
BEGIN
    -- wyświetla liczbę adresów w p_addresses
    DBMS_OUTPUT.PUT_LINE(
        'Bieżąca liczba adresów = ' || p_addresses.COUNT
    );

    -- korzystając z pętli, wyświetla adresy znajdujące się w p_addresses
    FOR v_count IN 1..p_addresses.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Adres nr ' || v_count || ':');
        DBMS_OUTPUT.PUT(p_addresses(v_count).street || ', ');
        DBMS_OUTPUT.PUT(p_addresses(v_count).city || ', ');
        DBMS_OUTPUT.PUT(p_addresses(v_count).state || ', ');
        DBMS_OUTPUT.PUT_LINE(p_addresses(v_count).zip);
    END LOOP;
END display_addresses;
```

Zastosowanie procedury `display_addresses()` zostanie wkrótce przedstawione.

Metoda DELETE()

Metoda `DELETE` usuwa elementy z kolekcji. Występują trzy formy tej metody:

- `DELETE` usuwa wszystkie elementy,
- `DELETE(n)` usuwa element *n*,
- `DELETE(n, m)` usuwa elementy od *n* do *m*.

Załóżmy, że mamy tabelę zagnieżdzoną o nazwie `v_nested_table`, która zawiera siedem elementów. W takiej sytuacji wywołanie `v_nested_table.DELETE(2, 5)` usunie elementy od 2 do 5.

Poniższa procedura `delete_address()` pobiera adresy klienta nr 1, a następnie, korzystając z metody `DELETE`, usuwa adres, którego indeks jest określany przez parametr `p_address_num`:

```
PROCEDURE delete_address(
    p_address_num INTEGER
) IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Usuwanie adresu nr ' || p_address_num);

    -- usuwa adres określony przez p_address_num
    v_addresses.DELETE(p_address_num);

    display_addresses(v_addresses);
END delete_address;
```

W poniższym przykładzie wywołano `delete_address(2)` w celu usunięcia adresu nr 2 klienta nr 1:

```
CALL collection_method_examples.delete_address(2);
Liczba adresów = 2
Bieżąca liczba adresów = 2
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:
Górska 4, Rzeszotary, GDA, 54321
Usuwanie adresu nr 2
Bieżąca liczba adresów = 1
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
```

Metoda EXISTS()

Metoda `EXISTS(n)` zwraca `true`, jeżeli w kolekcji istnieje element *n*. Metoda ta zwraca `true` dla niepustych elementów i zwraca `false` dla pustych elementów tabeli zagnieżdzonej lub elementów spoza zakresu kolekcji. Założymy, że mamy tabelę zagnieżdzoną o nazwie `v_nested_table`, która zawiera poniższy układ elementów:

Indeks elementu	Pusty/niepusty
1	pusty
2	niepusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie `v_nested_table.EXISTS(2)` zwróci `true` (ponieważ element nr 2 nie jest pusty), a wywołanie `v_nested_table.EXISTS(3)` zwróci `false` (ponieważ element nr 3 jest pusty).

Poniższa procedura `exists_addresses()` pobiera adresy klienta nr 1, usuwa adres nr 1 za pomocą metody `DELETE`, a następnie sprawdza za pomocą metody `EXISTS`, czy istnieją adresy nr 1 i 2 (nr 1 nie istnieje, bo został usunięty, istnieje tylko nr 2):

```

PROCEDURE exist_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    DBMS_OUTPUT.PUT_LINE('Usuwanie adresu nr 1');
    v_addresses.DELETE(1);

    -- używa EXISTS do sprawdzenia, czy adresy istnieją
    IF v_addresses.EXISTS(1) THEN
        DBMS_OUTPUT.PUT_LINE('Adres nr 1 istnieje');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Adres nr 1 nie istnieje');
    END IF;
    IF v_addresses.EXISTS(2) THEN
        DBMS_OUTPUT.PUT_LINE('Adres nr 2 istnieje');
    END IF;
END exist_addresses;

```

W poniższym przykładzie wywołano procedurę `exist_addresses`:

```

CALL collection_method_examples.exist_addresses();
Liczba adresów = 2
Usuwanie adresu nr 1
Adres nr 1 nie istnieje
Adres nr 2 istnieje

```

Metoda EXTEND()

Metoda `EXTEND()` dodaje elementy na koniec kolekcji. Występują trzy formy tej metody:

- `EXTEND` wstawia jeden element o wartości `NULL`,
- `EXTEND(n)` wstawia *n* elementów o wartości `NULL`,
- `EXTEND(n, m)` wstawia *n* elementów o wartości skopiowanej z elementu *m*.

Załóżmy, że mamy kolekcję o nazwie `v_nested_table`, zawierającą siedem elementów. W takiej sytuacji wywołanie `v_nested_table.EXTEND(2, 5)` dwukrotnie wstawi element nr 5 na końcu kolekcji.

Poniższa procedura `extend_addresses()` pobiera adresy klienta nr 1 do obiektu `v_address`, a następnie, korzystając z metody `EXTEND()`, kopiuje dwukrotnie adres nr 1 na koniec kolekcji:

```

PROCEDURE extend_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Rozszerzanie adresów');

    -- kopiuje dwukrotnie adres nr 1 na koniec v_addresses
    v_addresses.EXTEND(2, 1);

    display_addresses(v_addresses);
END extend_addresses;

```

Poniżej przedstawiono wynik wywołania procedury `extend_addresses()`:

```

CALL collection_method_examples.extend_addresses();
Liczba adresów = 2
Bieżąca liczba adresów = 2
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:
Górska 4, Rzeszotary, GDA, 54321
Rozszerzanie adresów
Bieżąca liczba adresów = 4
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:

```

Góriska 4, Rzeszotary, GDA, 54321

Adres nr 3:

Stanowa 2, Fasolowo, MAZ, 12345

Adres nr 4:

Stanowa 2, Fasolowo, MAZ, 12345

Metoda FIRST()

Metoda **FIRST()** zwraca indeks pierwszego elementu kolekcji. Jeżeli kolekcja jest pusta, metoda **FIRST()** zwraca **NULL**. Ponieważ pojedyncze elementy tabeli zagnieżdżonej mogą być puste, w przypadku tabeli zagnieżdżonej metoda **FIRST()** zwraca najniższy indeks niepustego elementu. Założymy, że mamy tabelę zagnieżdzoną o nazwie **v_nested_table** z następującym układem elementów:

Indeks elementu	Pusty/niepusty
1	pusty
2	niepusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie **v_nested_table.FIRST** zwróci 2, czyli najniższy indeks zawierający niepusty element.

Poniższa procedura **first_address()** pobiera adresy klienta nr 1 do obiektu **v_addresses**, a następnie za pomocą metody **FIRST()** wyświetla indeks pierwszego adresu z **v_addresses**. Procedura usuwa adres nr 1 za pomocą metody **DELETE** i wyświetla nowy indeks zwracany przez metodę **FIRST()**:

```
PROCEDURE first_address IS
  v_addresses t_nested_table_address;
BEGIN
  v_addresses := get_addresses(1);

  -- wyświetla adres zwracany przez FIRST
  DBMS_OUTPUT.PUT_LINE('Pierwszy adres = ' || v_addresses.FIRST);
  DBMS_OUTPUT.PUT_LINE('Usuwanie adresu nr 1');
  v_addresses.DELETE(1);

  -- ponownie wyświetla adres zwracany przez FIRST
  DBMS_OUTPUT.PUT_LINE('Pierwszy adres = ' || v_addresses.FIRST);
END first_address;
```

Poniżej przedstawiono wynik wywołania procedury **first_address()**:

```
CALL collection_method_examples.first_address();
Liczba adresów = 2
Pierwszy adres = 1
Usuwanie adresu nr 1
Pierwszy adres = 2
```

Metoda LAST()

Metoda **LAST()** zwraca indeks ostatniego elementu kolekcji. Jeżeli kolekcja jest pusta, metoda **LAST()** zwraca **NULL**. Ponieważ pojedyncze elementy tabeli zagnieżdżonej mogą być puste, w przypadku tabeli zagnieżdżonej metoda **LAST()** zwraca największy indeks niepustego elementu. Założymy, że mamy tabelę zagnieżdzoną o nazwie **v_nested_table** z następującym układem elementów:

Indeks elementu	Pusty/niepusty
1	niepusty
2	pusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie `v_nested_table.LAST` zwróci 4, czyli najwyższy indeks zawierający niepusty element.

Poniższa procedura `last_address()` pobiera adresy klienta nr 1 do obiektu `v_addresses`, a następnie za pomocą metody `LAST()` wyświetla indeks ostatniego adresu z `v_addresses`. Następnie procedura usuwa adres nr 2 za pomocą metody `DELETE` i wyświetla nowy indeks zwracany przez metodę `LAST()`:

```
PROCEDURE last_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- wyświetla adres zwracany przez LAST
    DBMS_OUTPUT.PUT_LINE('Ostatni adres = ' || v_addresses.LAST);
    DBMS_OUTPUT.PUT_LINE('Usuwanie adresu nr 2');
    v_addresses.DELETE(2);

    -- ponownie wyświetla adres zwracany przez LAST
    DBMS_OUTPUT.PUT_LINE('Ostatni adres = ' || v_addresses.LAST);
END last_address;
```

Poniżej przedstawiono wynik wywołania procedury `last_address()`:

```
CALL collection_method_examples.last_address();
Liczba adresów = 2
Ostatni adres = 2
Usuwanie adresu nr 2
Ostatni adres = 1
```

Metoda `NEXT()`

Metoda `NEXT(n)` zwraca element znajdujący się za elementem *n*. Ponieważ pojedyncze elementy tabeli zagnieżdżonej mogą być puste, metoda `NEXT()` zwraca indeks niepustego elementu za *n*. Jeżeli po *n* nie ma żadnych elementów, metoda `NEXT` zwraca `NULL`. Założymy, że mamy zagnieżdżoną tabelę o nazwie `v_nested_table` z następującym układem elementów:

Indeks elementu	Pusty/niepusty
1	niepusty
2	pusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie `v_nested_table.NEXT(1)` zwróci 4, czyli indeks zawierający kolejny niepusty element, a wywołanie `v_nested_value.NEXT(4)` zwróci `NULL`.

Poniższa procedura `next_address()` pobiera adresy klienta nr 1 do obiektu `v_addresses`, a następnie za pomocą wywołania `NEXT(1)` pobiera indeks adresu znajdującego się za adresem nr 1 w `v_addresses`. Procedura wywołuje `NEXT(2)`, próbując pobrać indeks adresu znajdującego się za adresem 2 (taki adres nie istnieje, ponieważ dla klienta nr 1 określono tylko dwa adresy, jest więc zwartana wartość `NULL`):

```
PROCEDURE next_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- użycie NEXT(1) do pobrania indeksu adresu
    -- znajdującego się za adresem nr 1
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.NEXT(1) = ' || v_addresses.NEXT(1)
    );

    -- użycie NEXT(2) w celu pobrania indeksu
    -- adresu znajdującego się za adresem nr 2
```

```
-- (taki adres nie istnieje, więc zwracane jest NULL)
DBMS_OUTPUT.PUT_LINE(
    'v_addresses.NEXT(2) = ' || v_addresses.NEXT(2)
);
END next_address;
```

Poniżej przedstawiono wynik wywołania metody `next_address()`. Wynik wywołania `v_addresses.NEXT(2)` jest wartością `NULL`, stąd brak wyniku za znakiem = dla tego elementu:

```
CALL collection_method_examples.next_address();
Liczba adresów = 2
v_addresses.NEXT(1) = 2
v_addresses.NEXT(2) =
```

Metoda PRIOR()

Metoda `PRIOR(n)` zwraca indeks elementu znajdującego się przed *n*. Ponieważ pojedyncze elementy tabeli zagnieżdżonej mogą być puste, metoda `PRIOR(n)` zwraca indeks niepustego elementu przed *n*. Jeżeli przed *n* nie ma żadnych elementów, metoda `PRIOR()` zwraca `NULL`. Założymy, że mamy zagnieżdżoną tabelę o nazwie `v_nested_table` z następującym układem elementów:

Indeks elementu	Pusty/niepusty
1	niepusty
2	pusty
3	pusty
4	niepusty

W takiej sytuacji wywołanie `v_nested_table.LAST(4)` zwróci 1, czyli indeks zawierający wcześniejszy niepusty element, wywołanie `v_nested_value.NEXT(1)` zwróci `NULL`.

Poniższa procedura `prior_address()` pobiera adresy klienta nr 1 do obiektu `v_addresses`, a następnie za pomocą wywołania `PRIOR(2)` pobiera indeks adresu znajdującego się przed adresem nr 2 w `v_addresses`. Procedura wywołuje `PRIOR(1)`, próbując pobrać indeks adresu znajdującego się przed adresem 1 (taki adres nie istnieje, jest więc zwracana wartość `NULL`):

```
PROCEDURE prior_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- użycie PRIOR(2) do pobrania indeksu adresu
    -- znajdującego się przed adresem nr 2
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.PRIOR(2) = ' || v_addresses.PRIOR(2)
    );

    -- użycie PRIOR(1) do pobrania indeksu
    -- adresu znajdującego się przed adresem nr 1
    -- (taki adres nie istnieje, więc zwracana jest wartość NULL)
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.PRIOR(1) = ' || v_addresses.PRIOR(1)
    );
END prior_address;
```

Poniższy przykład przedstawia wywołanie procedury `prior_address()`. Wynik wywołania `v_addresses.PRIOR(1)` jest wartością `NULL`, stąd brak wyniku za znakiem = dla tego elementu:

```
CALL collection_method_examples.prior_address();
Liczba adresów = 2
v_addresses.PRIOR(2) = 1
v_addresses.PRIOR(1) =
```

Metoda TRIM()

Metoda TRIM() usuwa elementy z końca kolekcji. Istnieją dwie formy tej metody:

- TRIM usuwa jeden element z końca kolekcji,
- TRIM(*n*) usuwa *n* elementów z końca kolekcji.

Załóżmy, że mamy tabelę zagnieżdżoną o nazwie v_nested_table — wywołanie v_nested_table.TRIM(2) usunie dwa elementy z końca tej tabeli.

Poniższa procedura trim_addresses() pobiera adresy klienta nr 1, kopiuje trzykrotnie adres nr 1 na koniec v_addresses, korzystając z EXTEND(3, 1), a następnie usuwa dwa adresy z końca v_addresses za pomocą wywołania TRIM(2):

```
PROCEDURE trim_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Rozszerzanie adresów');
    v_addresses.EXTEND(3, 1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Usuwanie 2 adresów z końca');

    -- usuwa 2 adresy z końca v_address,
    -- korzystając z TRIM(2)
    v_addresses.TRIM(2);

    display_addresses(v_addresses);
END trim_addresses;
```

Poniżej przedstawiono wynik wywołania procedury trim_addresses():

```
CALL collection_method_examples.trim_addresses();
Liczba adresów = 2
Bieżąca liczba adresów = 2
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:
Górska 4, Rzeszotary, GDA, 54321
Rozszerzanie adresów
Bieżąca liczba adresów = 5
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:
Górska 4, Rzeszotary, GDA, 54321
Adres nr 3:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 4:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 5:
Stanowa 2, Fasolowo, MAZ, 12345
Usuwanie 2 adresów z końca
Bieżąca liczba adresów = 3
Adres nr 1:
Stanowa 2, Fasolowo, MAZ, 12345
Adres nr 2:
Górska 4, Rzeszotary, GDA, 54321
Adres nr 3:
Stanowa 2, Fasolowo, MAZ, 12345
```

Kolekcje wielopoziomowe

W Oracle Database 9i wprowadzono możliwość tworzenia kolekcji, których elementami są inne kolekcje. Te „kolekcje kolekcji” nazywamy **kolekcjami wielopoziomowymi**. Poniżej wymieniono dopuszczalne kolekcje wielopoziomowe:

- tabela zagnieżdzona zawierająca tabele zagnieżdzone,
- tabela zagnieżdzona zawierająca tablice VARRAY,
- tablica VARRAY zawierająca tablice VARRAY,
- tablica VARRAY zawierająca tabele zagnieżdzone.

Następne podrozdziały pokazują, jak uruchomić skrypt tworzących przykładowy schemat danych i jak używać kolekcji wielopoziomowych.

Uruchomienie skryptu tworzącego schemat bazy danych collection_schema2

W katalogu SQL znajduje się skrypt *collection_schema2.sql*. Tworzy on konto użytkownika o nazwie *collection_user2* i hasło *collection_password*, a także wszystkie typy i tabele prezentowane w tym podrozdziale. Ten skrypt może być uruchomiony w Oracle Database 9i lub nowszej.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika **system**.
3. Uruchomić skrypt *collection_schema2.sql* w SQL*Plus za pomocą polecenia **@**.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu C:\SQL, to należy wpisać polecenie:

```
@ C:\SQL\collection_schema2.sql
```

Po zakończeniu pracy skryptu będzie zalogowany użytkownik *collection_user2*.

Korzystanie z kolekcji wielopoziomowych

Załóżmy, że dla każdego adresu klienta chcemy składować zbiór numerów telefonów. Poniższy przykład tworzy typ VARRAY o nazwie *t_varray_phone*, zawierający trzy napisy VARCHAR2. Ten typ będzie reprezentował numery telefonów:

```
CREATE TYPE t_varray_phone AS VARRAY(3) OF VARCHAR2(14);
/
```

Poniższy przykład tworzy typ obiektowy o nazwie *t_address*, który zawiera atrybut *phone_numbers*. Typem tego atrybutu jest *t_varray_phone*:

```
CREATE TYPE t_address AS OBJECT (
    street      VARCHAR2(15),
    city        VARCHAR2(15),
    state       CHAR(3),
    zip         VARCHAR2(5),
    phone_numbers t_varray_phone
);
/
```

Kolejny przykład tworzy tabelę zagnieżdzoną obiektów *t_address*:

```
CREATE TYPE t_nested_table_address AS TABLE OF t_address;
/
```

Poniższy przykład tworzy tabelę o nazwie *customers_with_nested_table*, która zawiera kolumnę *addresses* o typie *t_nested_table_address*:

```
CREATE TABLE customers_with_nested_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_nested_table_address
);
```

```
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses;
```

Tabela `customers_with_nested_table` zawiera więc tabelę zagnieźdzoną, której elementy zawierają z kolei adres z tablicą VARRAY, a w niej znajdują się numery telefonów.

Poniższa instrukcja `INSERT` wstawia wiersz do tabeli `customers_with_nested_table`. Należy zwrócić uwagę na strukturę i treść instrukcji `INSERT`, zawierającej elementy dla zagnieźdzonej tabeli z adresami — w każdym z tych elementów jest osadzona tablica VARRAY z numerami telefonów:

```
INSERT INTO customers_with_nested_table VALUES (
    1, 'Stefan', 'Brązowy',
    t_nested_table_address(
        t_address('Stanowa 2', 'Fasolowo', 'MAZ', '12345',
            t_varray_phone(
                '(22)-555-1211',
                '(22)-555-1212',
                '(22)-555-1213
            )
        ),
        t_address('Wysoka 4', 'Skomilice', 'MAŁ', '54321',
            t_varray_phone(
                '(12)-555-1211',
                '(12)-555-1212
            )
        )
    )
);
```

Widzimy, że do pierwszego adresu są przypisane trzy numery telefonów, a do drugiego adresu — dwa. Poniższe zapytanie pobiera wiersz z tabeli `customers_with_nested_table`:

```
SELECT *
FROM customers_with_nested_table;
```

```
ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP, PHONE_NUMBERS)
-----
1 Stefan     Brązowy
T_NESTED_TABLE_ADDRESS(
    T_ADDRESS('Stanowa 2', 'Fasolowo', 'MAZ', '12345',
        T_VARRAY_PHONE('(22)-555-1211', '(22)-555-1212', '(22)-555-1213')),
    T_ADDRESS('Wysoka 4', 'Skomilice', 'MAŁ', '54321',
        T_VARRAY_PHONE('(12)-555-1211', '(12)-555-1212')))
```

Możemy użyć funkcji `TABLE()` w celu zinterpretowania danych składowanych kolekcji jako serii wierszy, co obrazuje poniższe zapytanie:

```
SELECT cn.first_name, cn.last_name, a.street, a.city, a.state, p.*
FROM customers_with_nested_table cn,
    TABLE(cn.addresses) a, TABLE(a.phone_numbers) p;
```

FIRST_NAME	LAST_NAME	STREET	CITY	STATE	COLUMN_VALUE
Stefan	Brązowy	Stanowa 2	Fasolowo	MAZ	(22)-555-1211
Stefan	Brązowy	Stanowa 2	Fasolowo	MAZ	(22)-555-1212
Stefan	Brązowy	Stanowa 2	Fasolowo	MAZ	(22)-555-1213
Stefan	Brązowy	Wysoka 4	Skomilice	MAŁ	(12)-555-1211
Stefan	Brązowy	Wysoka 4	Skomilice	MAŁ	(12)-555-1212

Poniższa instrukcja `UPDATE` obrazuje, jak zmodyfikować numery telefonów przypisane do adresu Stanowa 2. Wykorzystano funkcję `TABLE()` w celu pobrania adresów jako serii wierszy, a w klauzuli `SET` została umieszczona tablica VARRAY, zawierająca nowe numery telefonów:

```

UPDATE TABLE(
    -- pobiera adresy klienta nr 1
    SELECT cn.addresses
    FROM customers_with_nested_table cn
    WHERE cn.id = 1
) addrs
SET addrs.phone_numbers =
    t_varray_phone(
        '(22)-555-1214',
        '(22)-555-1215'
    )
WHERE addrs.street = 'Stanowa 2';

```

1 row updated.

Kolejne zapytanie weryfikuje wprowadzenie zmian:

```

SELECT cn.first_name, cn.last_name, a.street, a.city, a.state, p.*
FROM customers_with_nested_table cn,
TABLE(cn.addresses) a, TABLE(a.phone_numbers) p;

```

FIRST_NAME	LAST_NAME	STREET	CITY	STATE	COLUMN_VALUE
Stefan	Brażowy	Stanowa 2	Fasolowo	MAZ	(22)-555-1214
Stefan	Brażowy	Stanowa 2	Fasolowo	MAZ	(22)-555-1215
Stefan	Brażowy	Wysoka 4	Skomilice	MAŁ	(12)-555-1211
Stefan	Brażowy	Wysoka 4	Skomilice	MAŁ	(12)-555-1212

Obsługa wielopoziomowych kolekcji przez bazę danych Oracle daje wiele możliwości i warto rozważyć ich zastosowanie w każdym projekcie bazy danych.

Rozszerzenia kolekcji wprowadzone w Oracle Database 10g

W tym podrozdziale poznasz następujące rozszerzenia kolekcji wprowadzone w Oracle Database 10g:

- obsługę tablic asocjacyjnych,
- możliwość zmiany rozmiaru lub precyzji typu elementu,
- możliwość zwiększenia liczby elementów w tablicy VARRAY,
- możliwość użycia kolumn typu VARRAY w tabelach tymczasowych,
- możliwość użycia innej przestrzeni tabel dla tabeli składającej się zagnieżdżonej tabeli zagnieżdżonej,
- obsługę tabel zagnieżdżonych realizowaną w standardzie ANSI.

Uruchomienie skryptu tworzącego schemat bazy danych collection_schema3

Instrukcje tworzące elementy prezentowane w tym podrozdziale zostały umieszczone w skrypcie *collection_schema3.sql*. Ten skrypt tworzy użytkownika o nazwie **collection_user3** i hasło **collection_password**. Tworzy on także typy kolekcji, tabele i kod PL/SQL. Może zostać uruchomiony w Oracle Database 10g lub nowszej.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika **system**.
3. Uruchomić skrypt *collection_schema3.sql* w SQL*Plus za pomocą polecenia **@**.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu C:\SQL, to należy wpisać polecenie:

```
@ C:\SQL\collection_schema3.sql
```

Po zakończeniu pracy skryptu będzie zalogowany użytkownik collection_user3.

Tablice asocjacyjne

Tablica asocjacyjna jest zbiorem par klucz-wartość. Wartość z tablicy można pobrać za pomocą klucza (który może być napisem) lub liczby całkowitej określającej pozycję wartości w tablicy. Poniższa, przykładowa procedura `customers_associative_array()` obrazuje użycie tablic asocjacyjnych:

```
CREATE OR REPLACE PROCEDURE customers_associative_array AS
  -- definiuje typ tablicy asocjacyjnej o nazwie t_assoc_array;
  -- wartość składowana w każdym elemencie tablicy jest typu NUMBER,
  -- a klucz indeksu do pobrania każdego elementu jest typu VARCHAR2
  TYPE t_assoc_array IS TABLE OF NUMBER INDEX BY VARCHAR2(15);

  -- deklaracja obiektu v_customer_array o typie t_assoc_array;
  -- v_customer_array będzie użyty do składowania wieku klientów
  v_customer_array t_assoc_array;
BEGIN
  -- przypisuje wartości v_customer_array; Klucz VARCHAR2
  -- jest nazwą klienta, a wartość NUMBER jest wiekiem klienta
  v_customer_array('Jan') := 32;
  v_customer_array('Stefan') := 28;
  v_customer_array('Fryderyk') := 43;
  v_customer_array('Sylwia') := 27;

  -- wyświetla wartości składowane w v_customer_array
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array["Jan"] = ' || v_customer_array('Jan')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array["Stefan"] = ' || v_customer_array('Stefan')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array["Fryderyk"] = ' || v_customer_array('Fryderyk')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array["Sylwia"] = ' || v_customer_array('Sylwia')
  );
END customers_associative_array;
/
```

Poniższy przykład włącza wypisywanie informacji z serwera oraz wywołuje procedurę `customers_associative_array()`:

```
SET SERVEROUTPUT ON
CALL customers_associative_array();
v_customer_array['Jan'] = 32
v_customer_array['Stefan'] = 28
v_customer_array['Fryderyk'] = 43
v_customer_array['Sylwia'] = 27
```

Zmienianie rozmiaru typu elementu

Rozmiar typu elementu w kolekcji można zmienić, jeżeli typ elementu jest jednym z typów znakowych, liczbowych lub surowych (typy surowe są wykorzystywane do składowania danych binarnych — informacje na ten temat znajdują się w kolejnym rozdziale). Wcześniej w tym rozdziale widzieliśmy poniższą instrukcję tworzącą typ VARRAY o nazwie `t_varray_address`:

```
CREATE TYPE t_varray_address AS VARRAY(3) OF VARCHAR2(50);
/
```

W poniższym przykładzie rozmiar elementów VARCHAR2 w t_varray_address jest zmieniany do 60 znaków:

```
ALTER TYPE t_varray_address
MODIFY ELEMENT TYPE VARCHAR2(60) CASCADE;
```

Type altered.

Opcja CASCADE powoduje propagację zmiany do wszystkich obiektów zależnych w bazie danych. W przykładzie takim obiektem jest tabela customers_with_varray, zawierająca kolumnę o nazwie addresses i typie t_varray_address.

Użycie opcji INVALIDATE powoduje natomiast unieważnienie obiektów zależnych i rekompilację kodu PL/SQL dla tego typu.

Zwiększenie liczby elementów w tablicy VARRAY

Można zwiększyć liczbę elementów w tablicy VARRAY. W poniższym przykładzie zwiększoano liczbę elementów w t_varray_address do pięciu:

```
ALTER TYPE t_varray_address
MODIFY LIMIT 5 CASCADE;
```

Type altered.

Użycie tablic VARRAY w tabelach tymczasowych

Tablice VARRAY mogą być używane w tabelach tymczasowych, związanych z określona sesją użytkownika (tabele tymczasowe zostały opisane w podrozdziale „Tworzenie tabeli” w rozdziale 11.) W poniższym przykładzie tworzona jest tabela tymczasowa o nazwie cust_with_varray_temp_table, zawierająca tablicę VARRAY o nazwie addresses i typie t_varray_address:

```
CREATE GLOBAL TEMPORARY TABLE cust_with_varray_temp_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_varray_address
);
```

Użycie innej przestrzeni tabel dla tabeli składającej tabelę zagnieżdzoną

Domyślnie tabela składowania tabeli zagnieżdzionej jest tworzona w tej samej przestrzeni tabel co tabela nadrzędna (przestrzeń tabel jest obszarem używanym przez bazę danych do składowania obiektów takich jak tabele — szczegółowe informacje na ten temat znajdują się w podrozdziale „Tworzenie tabeli” w rozdziale 11.).

W Oracle Database 10g i nowszych można określić inną przestrzeń tabel dla tabeli składowania tabeli zagnieżdzionej. Poniższy przykład tworzy tabelę cust_with_nested_table, zawierającą tabelę zagnieżdzoną o nazwie addresses i typie t_nested_table_address. Należy zauważyć, że przestrzenią tabel dla tabeli składowania nested_addresses2 jest przestrzeń users:

```
CREATE TABLE cust_with_nested_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_nested_table_address
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses2 TABLESPACE users;
```

Aby ten przykład zadziałał, w bazie danych musi istnieć przestrzeń tabel `users`. Z tego powodu w skrypcie `collection_schema3.sql` został on umieszczony w komentarzu (co zapobiega jego wykonaniu). Wszystkie dostępne przestrzenie tabel są zwracane przez następujące zapytanie:

```
SELECT tablespace_name
FROM user_tablespaces;

TABLESPACE_NAME
-----
SYSTEM
SYSAUX
UNDOTBS1
TEMP
USERS
EXAMPLE
```

W celu samodzielnego uruchomienia wcześniej przedstawionej instrukcji `CREATE TABLE` należy edytować przykład w skrypcie `collection_schema3.sql`, aby odwoływał się do jednej z dostępnych przestrzeni tabel, a następnie skopiować instrukcję do SQL*Plus i uruchomić ją.

Obsługa tabel zagnieżdżonych w standardzie ANSI

Specyfikacja ANSI zawiera szereg operatorów pracujących na tabelach zagnieżdżonych. Zostaną one opisane w tym podrozdziale.

Operatory równości i nierówności

Operatory równości (=) i nierówności (<>) porównują dwie tabele zagnieżdżone, które są uważane za równe, jeżeli:

- tabele są tego samego typu,
- tabele mają tę samą licznosć, czyli zawierają taką samą liczbę elementów,
- wszystkie elementy w tabelach mają tę samą wartość.

Poniższa procedura `equal_example()` ilustruje użycie operatorów równości i nierówności:

```
CREATE OR REPLACE PROCEDURE equal_example AS
-- deklaracja typu o nazwie t_nested_table
TYPE t_nested_table IS TABLE OF VARCHAR2(10);

-- tworzenie obiektów typu t_nested_table o nazwach
-- v_customer_nested_table1, v_customer_nested_table2 i
-- v_customer_nested_table1. Będą w nich składowane
-- nazwy klientów
v_customer_nested_table1 t_nested_table := 
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
v_customer_nested_table2 t_nested_table := 
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
v_customer_nested_table3 t_nested_table := 
    t_nested_table('Józef', 'Jerzy', 'Zuzanna');

v_result BOOLEAN;
BEGIN
-- użycie operatora = do porównania v_customer_nested_table1 z
-- v_customer_nested_table2 (zawierają te same nazwy,
-- więc v_result ma wartość true)
v_result := v_customer_nested_table1 = v_customer_nested_table2;
IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
        'v_customer_nested_table1 równe v_customer_nested_table2'
    );
END IF;
```

```
-- użycie operatora <> do porównania v_customer_nested_table1 z
-- v_customer_nested_table3 (nie są równe, ponieważ pierwsze nazwy,
-- 'Fryderyk' i 'Józef' są różne, więc v_result ma wartość false)
v_result := v_customer_nested_table1 <> v_customer_nested_table3;
IF v_result THEN
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_nested_table1 nie równe v_customer_nested_table3'
  );
END IF;
END equal_example;
/
```

Poniższy przykład prezentuje wywołanie procedury `equal_example()`:

```
CALL equal_example();
v_customer_nested_table1 równe v_customer_nested_table2
v_customer_nested_table1 nie równe v_customer_nested_table3
```

Operatory IN i NOT IN

Operator `IN` sprawdza, czy elementy jednej tabeli zagnieżdżonej występują w innej tabeli zagnieżdżonej. Operator `NOT IN` sprawdza natomiast, czy elementy jednej tabeli zagnieżdżonej nie występują w innej tabeli zagnieżdżonej. Poniższa procedura `in_example()` obrazuje użycie tych operatorów:

```
CREATE OR REPLACE PROCEDURE in_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_customer_nested_table2 t_nested_table :=
    t_nested_table('Józef', 'Jerzy', 'Zuzanna');
  v_customer_nested_table3 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_result BOOLEAN;
BEGIN
  -- Użycie operatora IN do sprawdzenia, czy elementy tabeli
  -- v_customer_nested_table3 występują w tabeli v_customer_nested_table1
  -- (występują, więc v_result ma wartość true)
  v_result := v_customer_nested_table3 IN
    (v_customer_nested_table1);
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table3 IN v_customer_nested_table1'
    );
  END IF;
  -- Użycie operatora NOT IN do sprawdzenia, czy elementy tabeli
  -- v_customer_nested_table3 występują w tabeli v_customer_nested_table2
  -- (nie występują, więc v_result ma wartość true)
  v_result := v_customer_nested_table3 NOT IN
    (v_customer_nested_table2);
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table3 NOT IN v_customer_nested_table2'
    );
  END IF;
END in_example;
/
```

Poniżej przedstawiono wynik wywołania procedury `in_example()`:

```
CALL in_example();
v_customer_nested_table3 IN v_customer_nested_table1
v_customer_nested_table3 NOT IN v_customer_nested_table2
```

Operator SUBMULTISET

Operator SUBMULTISET sprawdza, czy elementy jednej tabeli zagnieżdzonej stanowią podzbiór elementów innej tabeli zagnieżdzonej. Poniższa procedura `submultiset_example()` obrazuje użycie operatora SUBMULTISET:

```
CREATE OR REPLACE PROCEDURE submultiset_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table := 
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_customer_nested_table2 t_nested_table := 
    t_nested_table('Jerzy', 'Fryderyk', 'Zuzanna', 'Jan', 'Stefan');
  v_result BOOLEAN;
BEGIN
  -- użycie operatora SUBMULTISET do sprawdzenia, czy elementy tabeli
  -- v_customer_nested_table1 są podzbiorem tabeli v_customer_nested_table2
  -- (Tak jest, więc v_result ma wartość true).
  v_result := 
    v_customer_nested_table1 SUBMULTISET OF v_customer_nested_table2;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table1 jest podzbiorem v_customer_nested_table2'
    );
  END IF;
END submultiset_example;
/
```

Poniżej przedstawiono wynik wywołania procedury `submultiset_example()`:

```
CALL submultiset_example();
v_customer_nested_table1 jest podzbiorem v_customer_nested_table2
```

Operator MULTISET

Operator MULTISET zwraca tabelę zagnieżdzoną, której elementy mają wartości stanowiące określoną kombinację elementów z dwóch przesłanych tabel. Występują trzy odmiany operatora MULTISET:

- **MULTISET UNION** zwraca tabelę zagnieżdzoną, której elementy stanowią sumę elementów z przesłanych dwóch tabel,
- **MULTISET INTERSECT** zwraca tabelę zagnieżdzoną, w której znajdują się elementy wspólne w sprawdzanych tabelach zagnieżdzonych,
- **MULTISET EXCEPT** zwraca tabelę zagnieżdzoną, której elementami są tylko elementy występujące w pierwszej tabeli, lecz nie występujące w drugiej tabeli zagnieżdzonej.

Operator MULTISET może zostać również użyty z następującymi opcjami:

- **ALL** oznacza, że wszystkie stosowne elementy mają znaleźć się w zwracanej tabeli zagnieżdzonej. Jest to opcja domyślna. Na przykład `MULTISET UNION ALL` zwraca tabelę zagnieżdzoną, której elementami jest suma elementów z dwóch przesłanych tabel zagnieżdzonych. W zwracanej tabeli zagnieżdzonej znajdują się wszystkie elementy, w tym powtarzające się.
- **DISTINCT** oznacza, że w zwracanej tabeli zagnieżdzonej mają znaleźć się tylko niepowtarzające się elementy. Na przykład `MULTISET UNION DISTINCT` zwraca tabelę zagnieżdzoną, której elementami jest suma elementów dwóch przesłanych tabel zagnieżdzonych, ze zwracanej tabeli usuwane są jednak duplikaty.

Poniższa procedura `multiset_example()` obrazuje użycie operatora MULTISET:

```
CREATE OR REPLACE PROCEDURE multiset_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table := 
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_customer_nested_table2 t_nested_table := 
    t_nested_table('Jerzy', 'Stefan', 'Robert');
```

```

v_customer_nested_table3 t_nested_table;
v_count INTEGER;
BEGIN
-- użycie MULTISET UNION (zwraca tabelę zagnieżdzoną, której elementy stanowią sumę
-- elementów przesyłanych tabel zagnieżdzonych
v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISET UNION
        v_customer_nested_table2;
DBMS_OUTPUT.PUT('UNION: ');
FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- użycie MULTISET UNION DISTINCT (DISTINCT określa, że w zwracanej tabeli zagnieżdzonej
-- znajdują się jedynie niepowtarzające się elementy z dwóch tabel)
v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISET UNION DISTINCT
        v_customer_nested_table2;
DBMS_OUTPUT.PUT('UNION DISTINCT: ');
FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- użycie MULTISET INTERSECT (zwraca tabelę zagnieżdzoną, której elementami
-- są elementy wspólne w przesyłanych tabelach zagnieżdzonych
v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISET INTERSECT
        v_customer_nested_table2;
DBMS_OUTPUT.PUT('INTERSECT: ');
FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- użycie MULTISET EXCEPT (zwraca tabelę zagnieżdzoną, której elementy znajdują się
-- w pierwszej tabeli i nie znajdują się w drugiej tabeli
v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISET EXCEPT
        v_customer_nested_table2;
DBMS_OUTPUT.PUT_LINE('EXCEPT: ');
FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
END LOOP;
END multiset_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `multiset_example()`:

```

CALL multiset_example();
UNION: Fryderyk Jerzy Zuzanna Jerzy Stefan Robert
UNION DISTINCT: Fryderyk Jerzy Zuzanna Stefan Robert
INTERSECT: Jerzy
EXCEPT:

```

Funkcja CARDINALITY()

Funkcja `CARDINALITY()` zwraca liczbę elementów w kolekcji. Poniższa procedura `cardinality_example()` obrazuje użycie funkcji `CARDINALITY()`:

```

CREATE PROCEDURE cardinality_example AS
TYPE t_nested_table IS TABLE OF VARCHAR2(10);
v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
v_cardinality INTEGER;

```

```

BEGIN
  -- wywołanie CARDINALITY() w celu pobrania liczby elementów w
  -- v_customer_nested_table1
  v_cardinality := CARDINALITY(v_customer_nested_table1);
  DBMS_OUTPUT.PUT_LINE('v_cardinality = ' || v_cardinality);
END cardinality_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `cardinality_example()`:

```

CALL cardinality_example();
v_cardinality = 3

```

Operator MEMBER OF

Operator MEMBER OF sprawdza, czy element znajduje się w tabeli zagnieżdzonej. Poniższa procedura `member_of_example()` obrazuje zastosowanie tego operatora:

```

CREATE OR REPLACE PROCEDURE member_of_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_result BOOLEAN;
BEGIN
  -- użycie MEMBER OF do sprawdzenia, czy w v_customer_nested_table1
  -- znajduje się element 'Jerzy' (tak jest, więc v_result ma wartość true)
  v_result := 'Jerzy' MEMBER OF v_customer_nested_table1;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('"Jerzy" należy do zbioru');
  END IF;
END member_of_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `member_of_example()`:

```

CALL member_of_example();
"Jerzy" należy do zbioru

```

Funkcja SET()

Funkcja `SET()` rozpoczyna od przekształcenia tabeli zagnieżdzonej w zestaw, następnie usuwa duplikaty z tego zestawu i kończy pracę, zwracając zestaw jako tabelę zagnieżdzoną. Poniższa procedura `set_example()` obrazuje użycie tej funkcji:

```

CREATE OR REPLACE PROCEDURE set_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna', 'Jerzy');
  v_customer_nested_table2 t_nested_table;
  v_count INTEGER;
BEGIN
  -- wywołanie SET() w celu przekształcenia tabeli zagnieżdzonej
  -- w zestaw, usunięcia z niego duplikatów i pobrania
  -- zestawu jako tabeli zagnieżdzonej
  v_customer_nested_table2 := SET(v_customer_nested_table1);
  DBMS_OUTPUT.PUT('v_customer_nested_table2: ');
  FOR v_count IN 1..v_customer_nested_table2.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table2(v_count) || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('');
END set_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `set_example()`:

```

CALL set_example();
v_customer_nested_table2: Fryderyk Jerzy Zuzanna

```

Operator IS A SET

Operator IS A SET sprawdza, czy elementy tabeli zagnieżdzonej nie powtarzają się. Poniższa procedura `is_a_set_example()` obrazuje użycie operatora IS A SET:

```
CREATE OR REPLACE PROCEDURE is_a_set_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna', 'Jerzy');
  v_result BOOLEAN;
BEGIN
  -- użycie operatora IS A SET do sprawdzenia, czy elementy w
  -- v_customer_nested_table1 nie powtarzają się (powtarzają, więc
  -- v_result ma wartość false)
  v_result := v_customer_nested_table1 IS A SET;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('Wszystkie elementy są unikatowe');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Wśród elementów znajdują się duplikaty');
  END IF;
END is_a_set_example;
/
```

Poniżej przedstawiono wynik wywołania procedury `is_a_set_example()`:

```
CALL is_a_set_example();
Wśród elementów znajdują się duplikaty
```

Operator IS EMPTY

Operator IS EMPTY sprawdza, czy tabela zagnieżdzona nie zawiera elementów. Poniższa procedura `is_empty_example` obrazuje użycie tego operatora:

```
CREATE OR REPLACE PROCEDURE is_empty_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fryderyk', 'Jerzy', 'Zuzanna');
  v_result BOOLEAN;
BEGIN
  -- użycie operatora IS EMPTY do sprawdzenia, czy tabela
  -- v_customer_nested_table1 jest pusta (nie jest, więc
  -- v_result ma wartość false)
  v_result := v_customer_nested_table1 IS EMPTY;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('Tabela zagnieżdzona jest pusta');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Tabela zagnieżdzona zawiera elementy');
  END IF;
END is_empty_example;
/
```

Poniżej przedstawiono wynik wywołania procedury `is_empty_example()`:

```
CALL is_empty_example();
Tabela zagnieżdzona zawiera elementy
```

Funkcja COLLECT()

Funkcja `COLLECT()` zwraca tabelę zagnieżdzoną na podstawie zbioru elementów. Poniższe zapytanie obrazuje użycie funkcji `COLLECT()`:

```
SELECT COLLECT(first_name)
FROM customers_with_varray;
-----  
COLLECT(FIRST_NAME)
-----  
SYSTPVypnwSkSTZSp/mNi3ibkLQ==('Stefan', 'Jan')
```

Można również użyć funkcji `CAST()` do przekształcenia elementów zwróconych przez funkcję `COLLECT()` na określony typ, co obrazuje poniższy przykład:

```
SELECT CAST(COLLECT(first_name) AS t_table)
FROM customers_with_varray;
-----  
CAST(COLLECT(FIRST_NAME)AS T_TABLE)  
-----  
T_TABLE('Stefan', 'Jan')
```

Typ `t_table` użyty w powyższym przykładzie jest tworzony przez skrypt `collection_schema3.sql` za pomocą następującej instrukcji:

```
CREATE TYPE t_table AS TABLE OF VARCHAR2(10);
/
```

Funkcja POWERMULTISET()

Funkcja `POWERMULTISET()` zwraca wszystkie kombinacje elementów z danej tabeli zagnieżdzonej, co obrazuje poniższe zapytanie:

```
SELECT *
FROM TABLE(
    POWERMULTISET(t_table('To', 'jest', 'nasz', 'test'))
);
-----  
COLUMN_VALUE  
-----  
T_TABLE('To')  
T_TABLE('jest')  
T_TABLE('To', 'jest')  
T_TABLE('nasz')  
T_TABLE('To', 'nasz')  
T_TABLE('jest', 'nasz')  
T_TABLE('To', 'jest', 'nasz')  
T_TABLE('test')  
T_TABLE('To', 'test')  
T_TABLE('jest', 'test')  
T_TABLE('To', 'jest', 'test')  
T_TABLE('nasz', 'test')  
T_TABLE('To', 'nasz', 'test')  
T_TABLE('jest', 'nasz', 'test')  
T_TABLE('To', 'jest', 'nasz', 'test')
```

Funkcja POWERMULTISET_BY_CARDINALITY()

Funkcja `POWERMULTISET_BY_CARDINALITY()` zwraca kombinacje elementów danej tabeli zagnieżdzonej. Liczba (czyli liczność) elementów w kombinacji jest określona. Poniższe zapytanie obrazuje użycie tej funkcji. Liczność określono jako 3:

```
SELECT *
FROM TABLE(
    POWERMULTISET_BY_CARDINALITY(
        t_table('To', 'jest', 'nasz', 'test'), 3
    )
);
-----  
COLUMN_VALUE  
-----  
T_TABLE('To', 'jest', 'nasz')  
T_TABLE('To', 'jest', 'test')  
T_TABLE('To', 'nasz', 'test')  
T_TABLE('jest', 'nasz', 'test')
```

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- kolekcje umożliwiają składowanie zbiorów elementów,
- istnieją trzy typy kolekcji: tablice VARRAY, tabele zagnieżdżone i tablice asocjacyjne,
- w tablicy VARRAY można składować uporządkowany zbiór elementów — elementy te są tego samego typu, a tablica ma jeden wymiar; jej maksymalny rozmiar określany jest przy jej tworzeniu, można go jednak później zmienić,
- tabela zagnieżdżona jest tabelą osadzoną w innej tabeli — można do niej wstawiać pojedyncze elementy oraz modyfikować je i usuwać; nie posiada ona rozmiaru maksymalnego i można w niej składować dowolną liczbę elementów,
- tablica asocjacyjna jest zbiorem par klucz-wartość; wartość z tablicy asocjacyjnej można pobrać za pomocą klucza (którym może być napis) lub liczby całkowitej określającej pozycję wartości w tablicy,
- w kolekcjach można osadzać inne kolekcje — taką kolekcję nazywamy wówczas wielopoziomową.

W następnym rozdziale zostaną opisane duże obiekty.

ROZDZIAŁ

15

Duże obiekty

W tym rozdziale:

- dowiesz się, co to są duże obiekty (LOB),
- przejrzymy pliki, których zawartość zostanie użyta do umieszczenia danych w przykładowych obiektach LOB,
- poznasz różnice między poszczególnymi typami dużych obiektów,
- utworzymy tabele zawierające duże obiekty,
- użyjemy obiektów LOB w SQL i PL/SQL,
- poznasz typy LONG i LONG RAW,
- dowiesz się czegoś o rozszerzeniach LOB.

Podstawowe informacje o dużych obiektach (LOB)

Obecnie witryny internetowe często potrzebują plików multimedialnych. Od baz danych jest więc wymagana możliwość składowania takich elementów jak muzyka i filmy. Przed opublikowaniem Oracle Database 8 duże bloki danych znakowych musiały być składowane z użyciem typu danych LONG, a duże bloki danych binarnych wymagały składowania z użyciem typu LONG RAW lub krótszego RAW.

Wraz z wprowadzeniem Oracle Database 8 udostępniono nową klasę typów danych, zwanych **dużymi obiektami** (LOB). Typy LOB mogą zostać użyte do składowania danych binarnych, znakowych, a także odwołań do plików. Dane binarne mogą zawierać obrazy, muzykę, filmy, dokumenty, pliki wykonywalne itd. W zależności od konfiguracji bazy danych duże obiekty mogą pomieścić do 128 terabajtów danych.

Przykładowe pliki

W tym rozdziale będą wykorzystywane dwa pliki:

- *textContent.txt* — plik tekstowy,
- *binaryContent.doc* — dokument Microsoft Word.



Te pliki znajdują się w katalogu *pliki*, który został utworzony po rozpakowaniu archiwum z przykładowymi plikami. Jeżeli chcesz wykonywać przykłady samodzielnie, powinieneś skopiować katalog *pliki* na partycję C serwera bazy danych. Jeżeli korzystasz z systemu Linux lub Unix, katalog można skopiować na którąś z osobistych partycji.

Plik *textContent.txt* zawiera fragment *Makbeta*¹. Poniższy tekst jest wygłaszanego przez Makbeta na krótko przed śmiercią:

Ciągle to jutro, jutro i znów jutro
 Wije się w ciasnym kółku od dnia do dnia
 Aż do ostatniej głoski czasokresu;
 A wszystkie wczoraj to były pochodnie,
 Które głupciose naszej przyświecały
 W drodze do śmierci. Zgaśnij, wątłe światło!
 Życie jest tylko przechodnim półcieniem,
 Nędznym aktorem, który swoją rolę
 Przez parę godzin wygrawszy na scenie
 W nicość przepada — powieścią idioty,
 Głośną, wrzaskliwą, a nic nie znaczącą.

Plik *binaryContent.doc* jest dokumentem programu Microsoft Word, zawierającym ten sam tekst. (Dokument Worda jest plikiem binarnym). Choć w przykładach wykorzystano dokument Worda, można użyć dowolnego pliku binarnego, takiego jak MP3, DivX, JPEG, MPEG, PDF czy też EXE. Przykłady zostały przetestowane ze wszystkimi wymienionymi typami plików.

Rodzaje dużych obiektów

Wyróżniamy cztery rodzaje LOB:

- **CLOB** — znakowy typ LOB, używany do składowania danych znakowych.
- **NCLOB** — typ LOB z obsługą narodowego zestawu znaków, używany do składowania wielobajtowych danych znakowych (zwykle stosowanych dla znaków innych niż angielskie). Informacje na temat tych znaków można znaleźć w podręczniku *Oracle Database Globalization Support Guide* opublikowanym przez Oracle Corporation.
- **BLOB** — binarny typ LOB, używany do składowania danych binarnych.
- **BFILE** — typ binarny FILE, używany do składowania wskaźnika do pliku. Plik może znajdować się na dysku twardym, płycie CD, DVD, Blu-Ray, HD-DVD lub jakimkolwiek innym urządzeniu, do którego można uzyskać dostęp za pośrednictwem systemu plików serwera bazy danych. Sam plik nie jest składowany w bazie danych — jest składowany jedynie wskaźnik do pliku.

Przed opublikowaniem Oracle Database 8 do składowania dużych ilości danych mogły być wykorzystywane jedynie typy **LONG** i **LONG RAW**. Typy LOB mają kilka zalet w porównaniu do tych starszych typów:

- W typie LOB może być składowanych do 128 terabajtów danych. Jest to znacznie więcej, niż można składować w typach **LONG** i **LONG RAW**, dla których limitem są 2 gigabajty.
- Tabela może zawierać wiele kolumn typu LOB, ale tylko jedną **LONG** i **LONG RAW**.
- Dostęp do danych LOB może być uzyskiwany losowo, natomiast po dane **LONG** i **LONG RAW** można sięgać jedynie sekwencyjnie.

LOB składa się z dwóch części:

- **lokalizatora LOB**, czyli wskaźnika określającego umiejscowienie danych LOB,
- **danych LOB**, czyli faktycznych danych znakowych lub binarnych składowanych w LOB.

W zależności od ilości danych składowanych w kolumnie **CLOB**, **NCLOB** lub **BLOB** dane będą składowane w tabeli albo poza nią. Jeżeli danych jest mniej niż 4 kilobajty, będą składowane w tej samej tabeli; w przeciwnym razie będą składowane poza tabelą. W przypadku kolumny **BFILE** w tabeli jest składowany jedynie lokalizator wskazujący na zewnętrzny plik składowany w systemie plików.

¹ W tłumaczeniu Józefa Paczkowskiego — przyp. tłum.

Tworzenie tabel zawierających duże obiekty

W tym podrozdziale będziemy wykorzystywali trzy tabele:

- `clob_content` zawierającą kolumnę CLOB o nazwie `clob_column`,
- `blob_content` zawierającą kolumnę BLOB o nazwie `blob_column`,
- `bfile_content` zawierającą kolumnę BFILE o nazwie `bfile_column`.

W katalogu *SQL* znajduje się skrypt SQL*Plus o nazwie `lob_schema.sql`. Może on zostać uruchomiony w Oracle Database 8 i nowszych. Zawartość skryptu można podejrzeć za pomocą dowolnego edytora tekstowego.

Poniższa instrukcja skryptu tworzy obiekt katalogu nazwany `SAMPLE_FILES_DIR` w katalogu widocznym w systemie plików systemu Windows jako `C:\pliki`:

```
CREATE DIRECTORY SAMPLE_FILES_DIR AS 'C:\pliki';
```

Jeśli katalog *pliki* nie jest umieszczony w głównym katalogu partycji C, należy w powyższej instrukcji odpowiednio zmodyfikować ścieżkę dostępu.

Jeżeli używasz systemu Linux lub Unix, również należy odpowiednio zmodyfikować ścieżkę dostępu do katalogu. Przykładowo może ona mieć postać:

```
CREATE DIRECTORY SAMPLE_FILES_DIR AS '/tmp/pliki';
```

Po wprowadzeniu niezbędnych modyfikacji zapisz skrypt.

Należy się upewnić, że:

- Katalog *pliki* istnieje w systemie plików.
- Konto użytkownika systemu operacyjnego użyte do instalacji bazy danych Oracle ma uprawnienia zapisu i odczytu w katalogu *pliki* oraz do plików znajdujących się w tym katalogu.

Jeżeli korzystasz z systemu Windows, nie musisz przejmować się drugim punktem. Baza danych Oracle została prawdopodobnie zainstalowana za pomocą konta użytkownika mającego uprawnienia administratora, a tego typu konto ma uprawnienia do odczytu w całym systemie plików.

Jeśli korzystasz z systemu Linux lub Unix, będziesz musiał nadać uprawnienia do odczytu i zapisu w katalogu *pliki* i do plików wewnętrz tego katalogu odpowiedniemu kontu użytkownika Oracle, który jest właścicielem bazy danych. Takie uprawnienia przydziela się za pomocą polecenia `chmod`.



Uwaga Jeżeli podczas wykonywania przykładów z tego rozdziału pojawią się błędy związane z operacjami na plikach, będą one prawdopodobnie efektem nieprzydzielenia odpowiednich uprawnień.

Skrypt `lob_schema.sql` tworzy konto użytkownika o nazwie `lob_user` i hasło `lob_password`, a także wszystkie tabele i kod PL/SQL używane w pierwszej części tego rozdziału. Po zakończeniu pracy skryptu logowany jest użytkownik `lob_user`.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika `system`.
3. Uruchomić skrypt `lob_schema.sql` w SQL*Plus za pomocą polecenia `@`.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu `C:\SQL`, to należy wpisać polecenie:

```
@ C:\SQL\lob_schema.sql
```

Po zakończeniu pracy skryptu będzie zalogowany użytkownik `lob_user`.

Wymienione tabele są tworzone przez skrypt za pomocą następujących instrukcji:

```

CREATE TABLE clob_content (
    id          INTEGER PRIMARY KEY,
    clob_column CLOB NOT NULL
);

CREATE TABLE blob_content (
    id          INTEGER PRIMARY KEY,
    blob_column BLOB NOT NULL
);

CREATE TABLE bfile_content (
    id          INTEGER PRIMARY KEY,
    bfile_column BFILE NOT NULL
);

```

Użycie dużych obiektów w SQL

Z tego podrozdziału dowiesz się, w jaki sposób można manipulować dużymi obiektami za pomocą języka SQL. Rozpoczniemy od obiektów CLOB i BLOB, a następnie przejdziemy do obiektów BFILE.

Użycie obiektów CLOB i BLOB

W kolejnych podrozdziałach opisano, jak wypełnić danymi obiekty CLOB i BLOB, pobrać z nich dane i jak je zmodyfikować.

Umieszczanie danych w obiektach CLOB i BLOB

Poniższe instrukcje INSERT wstawiają dwa wiersze do tabeli `clob_content`. Należy zauważyć użycie funkcji `TO_CLOB()` w celu konwersji tekstu na CLOB:

```

INSERT INTO clob_content (
    id, clob_column
) VALUES (
    1, TO_CLOB('Wije się w ciasnym kółku')
);

INSERT INTO clob_content (
    id, clob_column
) VALUES (
    2, TO_CLOB(' od dnia do dnia ')
);

```

Poniższe instrukcje INSERT wstawiają dwa wiersze do tabeli `blob_content`. Należy zauważyć użycie funkcji `TO_BLOB()` w celu konwersji liczb na typ BLOB (pierwsza instrukcja zawiera liczbę w formacie binarnym, a druga instrukcja zawiera liczbę w formacie szesnastkowym):

```

INSERT INTO blob_content (
    id, blob_column
) VALUES (
    1, TO_BLOB('100111010101011111')
);

INSERT INTO blob_content (
    id, blob_column
) VALUES (
    2, TO_BLOB('A0FFB71CF90DE')
);

```

Pobieranie danych z CLOB

Poniższe zapytanie pobiera wiersze z tabeli `clob_content`:

```

SELECT *
FROM clob_content;

```

```

ID
-----
CLOB_COLUMN
-----
      1
Wije się w ciasnym kółku

      2
od dnia do dnia

Kolejne zapytanie próbuje pobrać wiersze z tabeli blob_content:

SELECT *
FROM blob_content;

ID
-----
BLOB_COLUMN
-----
      1
100111010101011111

      2
0A0FFB71CF90DE

```

Modyfikowanie danych w obiektach CLOB i BLOB

Instrukcje UPDATE i INSERT, prezentowane w tym podrozdziale, można uruchomić samodzielnie. Poniższe instrukcje UPDATE demonstrują modyfikowanie zawartości obiektów CLOB i BLOB:

```

UPDATE clob_content
SET clob_column = TO_CLOB('Co za blask strzelił tam z okna!')
WHERE id = 1;

UPDATE blob_content
SET blob_column = TO_BLOB('1110011010101011111')
WHERE id = 1;

```

Można również zainicjalizować lokalizator LOB, nie zapisując jednak rzeczywistych danych w LOB. Służą do tego funkcje EMPTY_CLOB() i EMPTY_BLOB():

```

INSERT INTO clob_content(
    id, clob_column
) VALUES (
    3, EMPTY_CLOB()
);

INSERT INTO blob_content(
    id, blob_column
) VALUES (
    3, EMPTY_BLOB()
);

```

Te instrukcje inicjalizują lokalizator LOB, ale obiekty LOB pozostają puste.

Funkcje EMPTY_CLOB() i EMPTY_BLOB() mogą zostać użyte w instrukcji UPDATE w celu usunięcia danych LOB. Na przykład:

```

UPDATE clob_content
SET clob_column = EMPTY_CLOB()
WHERE id = 1;

UPDATE blob_content
SET blob_column = EMPTY_BLOB()
WHERE id = 1;

```

Jeżeli uruchomiono prezentowane wyżej instrukcje INSERT i UPDATE, należy wycofać zmiany, aby dalsze wyniki były zgodne z prezentowanymi w książce:

ROLLBACK;

Użycie obiektów BFILE

Obiekt BFILE przechowuje wskaźnik do pliku, który jest dostępny za pośrednictwem systemu plików serwera bazy danych. Należy pamiętać, że te pliki są składowane poza bazą danych. BFILE może wskazywać na plik znajdujący się na dowolnym nośniku: dysku twardym, płycie CD, DVD, Blu-Ray, HD-DVD itd.

 Obiekt BFILE zawiera wskaźnik do pliku zewnętrznego. Rzeczywisty plik nie jest składowany w bazie danych i musi być osiągalny za pośrednictwem systemu plików serwera bazy danych.

Tworzenie obiektu katalogu

Zanim będzie możliwe zapisanie wskaźnika do pliku w obiekcie BFILE, konieczne jest utworzenie obiektu katalogu w bazie danych. Jest w nim składowany katalog systemu plików, w którym znajdują się pliki. Do tworzenia obiektu katalogu służy instrukcja CREATE DIRECTORY, do wydania której konieczne jest posiadanie uprawnienia CREATE ANY DIRECTORY.

Poniższy przykład (znajdujący się w skrypcie *lob_schema.sql*) tworzy obiekt katalogu o nazwie SAMPLE_FILES_DIR dla katalogu C:\sample_files z systemu plików:

```
CREATE DIRECTORY SAMPLE_FILES_DIR AS 'C:\sample_files';
```

Kolejny przykład przydziela uprawnienia do odczytu i zapisu w SAMPLE_FILES_DIR użytkownikowi lob_user:

```
GRANT read, write ON DIRECTORY SAMPLE_FILES_DIR TO lob_user;
```

Umieszczanie wskaźnika do pliku w obiekcie BFILE

Ponieważ BFILE jest jedynie wskaźnikiem do zewnętrznego pliku, umieszczanie danych w kolumnie tego typu jest bardzo proste. Wystarczy użyć funkcji BFILENAME() do umieszczenia w obiekcie BFILE wskaźnika do pliku zewnętrznego. Funkcja ta przyjmuje dwa parametry: nazwę obiektu katalogu oraz nazwę pliku.

Na przykład poniższa instrukcja INSERT wstawia wiersz do tabeli *bfile_content*. Należy zauważyć, że korzystając z funkcji BFILENAME(), w kolumnie *bfile_column* umieszczono wskaźnik do pliku *textContent.txt*:

```
INSERT INTO bfile_content (
    id, bfile_column
) VALUES (
    1, BFILENAME('SAMPLE_FILES_DIR', 'textContent.txt')
);
```

Kolejna instrukcja INSERT wstawia wiersz do tabeli *bfile_content*. Funkcja BFILENAME() umieszcza w kolumnie *bfile_column* wskaźnik do pliku *binaryContent.doc*:

```
INSERT INTO bfile_content (
    id, bfile_column
) VALUES (
    2, BFILENAME('SAMPLE_FILES_DIR', 'binaryContent.doc')
);
```

Poniższe zapytanie pobiera wiersze z tabeli *bfile_content*:

```
SELECT *
FROM bfile_content;
```

ID	-----	BFILER_COLUMN	-----

```

1
bfilename('SAMPLE_FILES_DIR', 'textContent.txt')

2
bfilename('SAMPLE_FILES_DIR', 'binaryContent.doc')

```

Dostęp do zawartości BFILE i BLOB można uzyskać za pomocą PL/SQL, co zostało opisane w kolejnym podrozdziale.

Użycie dużych obiektów w PL/SQL

W tym podrozdziale dowiesz się, jak używać obiektów LOB w PL/SQL. Rozpoczniemy od omówienia metod dostępnych w pakiecie DBMS_LOB, dostarczonym z bazą danych. Następnie zostanie przedstawionych wiele programów PL/SQL ilustrujących zastosowanie metod pakietu DBMS_LOB do odczytu danych z LOB, kopiowania danych z jednego LOB do innego, kopiowania danych z pliku do obiektu LOB, kopiowania danych z obiektu LOB do pliku i nie tylko.

W tabeli 15.1 opisano najczęściej używane metody z pakietu DBMS_LOB.

Tabela 15.1. Metody z pakietu DBMS_LOB

Metoda	Opis
APPEND(<i>lob_źródł</i> , <i>lob_doc</i>)	Umieszcza na końcu obiektu <i>lob_doc</i> dane odczytane z obiektu <i>lob_źródł</i>
CLOSE(<i>lob</i>)	Zamyka wcześniej otwarty obiekt LOB
COMPARE(<i>lob1</i> , <i>lob2</i> , <i>ilość</i> , <i>przesunięcie1</i> , <i>przesunięcie2</i>)	Porównuje dane składowane w <i>lob1</i> i <i>lob2</i> , rozpoczynając od <i>przesunięcie1</i> w <i>lob1</i> i <i>przesunięcie2</i> w <i>lob2</i> . Przesunięcia zawsze rozpoczynają się od 1, czyli pierwszego znaku lub bajta danych
CONVERTTOBLOB(<i>blob_doc</i> , <i>clob_źródł</i> , <i>ilość</i> , <i>przesunięcie_doc</i> , <i>przesunięcie_źródł</i> , <i>blob_csid</i> , <i>kontekst_językowy</i> , <i>ostrzeżenie</i>)	Porównywana jest zawsze określona liczba znaków lub bajtów (maksimum jest definiowane przez parametr <i>ilość</i>) Konwertuje dane odczytane z <i>clob_źródł</i> na dane binarne zapisywane do <i>blob_doc</i> Odczyt rozpoczyna się od <i>przesunięcie_źródł</i> w <i>clob_źródł</i> , a zapis w <i>przesunięcie_doc</i> w <i>blob_doc</i> <i>blob_csid</i> jest żądanym zestawem znaków dla konwertowanych danych zapisywanych do <i>blob_doc</i> . Zwykle należy stosować DMBS_LOB.DEFAULT.CSID, czyli domyślny zestaw znaków bazy danych <i>kontekst_językowy</i> jest kontekstem językowym używanym przy konwersji znaków odczytywanych z <i>clob_źródł</i> . Zwykle należy stosować DMBS_LOB.DEFAULT.LANG_CTX, czyli domyślny kontekst językowy bazy danych <i>ostrzeżenie</i> jest stawiane na DMBS_LOB.WARN_INCONVERTIBLE_CHAR, jeżeli napotkano znak, którego konwersja nie powiodła się
CONVERTTOCLOB(<i>clob_doc</i> , <i>blob_źródł</i> , <i>ilość</i> , <i>przesunięcie_doc</i> , <i>przesunięcie_źródł</i> , <i>blob_csid</i> , <i>kontekst_językowy</i> , <i>ostrzeżenie</i>)	Konwertuje dane binarne odczytane z <i>blob_źródł</i> na dane znakowe zapisywane do <i>clob_doc</i> <i>blob_csid</i> jest zestawem znaków ustalonym dla konwertowanych danych odczytywanych z <i>blob_źródł</i> . Zwykle należy stosować DMBS_LOB.DEFAULT.CSID, czyli domyślny zestaw znaków bazy danych <i>kontekst_językowy</i> jest kontekstem językowym używanym przy zapisie konwertowanych znaków do <i>clob_doc</i> . Zwykle należy stosować DMBS_LOB.DEFAULT.LANG_CTX <i>ostrzeżenie</i> jest stawiane na DMBS_LOB.WARN_INCONVERTIBLE_CHAR, jeżeli napotkano znak, którego konwersja nie powiodła się
COPY(<i>lob_doc</i> , <i>lob_źródł</i> , <i>ilość</i> , <i>przesunięcie_doc</i> , <i>przesunięcie_źródł</i>)	Kopiuje <i>ilość</i> znaków lub bajtów z <i>lob_źródł</i> do <i>lob_doc</i> , rozpoczynając w przesunięciach

Tabela 15.1. Metody z pakietu DBMS_LOB — ciąg dalszy

Metoda	Opis
CREATETEMPORARY(<i>lob, cache, czas</i>)	Tworzy tymczasowy LOB w domyślnej tymczasowej przestrzeni tabel użytkownika
ERASE(<i>lob, ilość, przesunięcie</i>)	Usuwa <i>ilosc</i> znaków lub bajtów z obiektu LOB, rozpoczynając od <i>przesunięcie</i>
FILECLOSE(<i>bfile</i>)	Zamyka <i>bfile</i> . Należy używać nowszej metody CLOSE() zamiast FILECLOSE()
FILECLOSEALL()	Zamyka wszystkie otwarte BFILE
FILEEXISTS(<i>bfile</i>)	Sprawdza, czy zewnętrzny plik wskazywany przez <i>bfile</i> faktycznie istnieje
FILEGETNAME(<i>bfile, katalog, nazwa_pliku</i>)	Zwraca katalog i nazwę zewnętrznego pliku, na który wskazuje <i>bfile</i>
FILEISOPEN(<i>bfile</i>)	Sprawdza, czy <i>bfile</i> jest otwarty. Zamiast tej metody należy używać nowszej, ISOPEN()
FILEOPEN(<i>bfile, tryb</i>)	Otwiera <i>bfile</i> w określonym trybie, którym może być jedynie DBMS_LOB.FILE_READONLY, czyli tylko do odczytu (zapis do pliku nie będzie możliwy). Zamiast FILEOPEN() należy używać nowszej metody OPEN()
FREETEMPORARY(<i>lob</i>)	Zwalnia tymczasowy obiekt LOB
GETCHUNKSIZE(<i>lob</i>)	Zwraca rozmiar fragmentu używanego przy odczytce i zapisie danych składowanych w LOB. Fragment jest jednostką danych
GET_STORAGE_LIMIT()	Zwraca maksymalny dozwolony rozmiar obiektu LOB
GETLENGTH(<i>lob</i>)	Pobiera długość danych składowanych w obiekcie LOB
INSTR(<i>lob, wzorzec, przesunięcie, n</i>)	Zwraca początkową pozycję znaków lub bajtów zgodnych z <i>n</i> -tym wystąpieniem wzorca w danych LOB. Punkt rozpoczęcia odczytu danych z LOB jest określany przez <i>przesunięcie</i>
ISOPEN(<i>lob</i>)	Sprawdza, czy obiekt LOB jest otwarty
ISTEMPORARY(<i>lob</i>)	Sprawdza, czy obiekt LOB jest obiektem tymczasowym
LOADFROMFILE(<i>lob_doc, bfile_źródł, ilość, przesunięcie_doc, przesunięcie_źródł</i>)	Wczytuje do <i>lob_doc</i> określoną <i>ilosc</i> znaków lub bajtów pobranych za pośrednictwem <i>bfile_źródł</i> . Odczyt i zapis jest rozpoczynany w przesunięciach. <i>bfile_źródł</i> jest obiektem BFILE wskazującym na plik zewnętrzny Funkcja LOADFROMFILE() jest przestarzała i należy używać nowszych, bardziej wydajnych metod, takich jak LOADBLOBFROMFILE() lub LOADCLOBFROMFILE()
LOADBLOBFROMFILE(<i>blob_doc, bfile_źródł, ilość, przesunięcie_doc, przesunięcie_źródł</i>)	Rozpoczynając w przesunięciach, wczytuje do <i>blob_doc</i> określoną <i>ilosc</i> bajtów odczytanych za pośrednictwem <i>bfile_źródł</i> . <i>bfile_źródł</i> jest obiektem BFILE wskazującym na plik zewnętrzny Przy pracy z obiektami BLOB metoda LOADBLOBFROMFILE() jest wydajniejsza niż LOADFROMFILE()
LOADCLOBFROMFILE(<i>clob_doc, bfile_źródł, ilość, przesunięcie_doc, przesunięcie_źródł, csid_źródł, kontekst_językowy, ostrzeżenie</i>)	Rozpoczynając w przesunięciach, wczytuje do <i>clob_doc</i> określoną <i>ilosc</i> znaków odczytanych za pośrednictwem <i>bfile_źródł</i> . <i>bfile_źródł</i> jest obiektem BFILE wskazującym na plik zewnętrzny Przy pracy z obiektami CLOB i NCLOB metoda LOADCLOBFROMFILE() jest wydajniejsza niż LOADFROMFILE()
LOBMAXSIZE	Zwraca maksymalny rozmiar obiektu LOB w bajtach (obecnie 2 ⁶⁴)
OPEN(<i>lob, tryb</i>)	Otwiera LOB we wskazanym trybie: <ul style="list-style-type: none"> ■ DBMS_LOB.FILE_READONLY dopuszcza jedynie odczyt z LOB ■ DBMS_LOB.FILE_READWRITE dopuszcza zarówno odczyt, jak i zapis do LOB

Tabela 15.1. Metody z pakietu DBMS_LOB — ciąg dalszy

Metoda	Opis
READ(<i>lob</i> , <i>ilość</i> , <i>przesunięcie</i> , <i>bufor</i>)	Odczytuje dane z LOB i zapisuje je do zmiennej <i>bufor</i> . Odczytywana jest określona <i>ilość</i> znaków lub bajtów, począwszy od <i>przesunięcia</i>
SUBSTR(<i>lob</i> , <i>ilość</i> , <i>przesunięcie</i>)	Zwraca część danych LOB, rozpoczynającą się od <i>przesunięcia</i> i obejmującą określona <i>ilość</i> bajtów lub znaków
TRIM(<i>lob</i> , <i>nowa_długość</i>)	Obcina dane LOB do określonej, nowej długości
WRITE(<i>lob</i> , <i>ilość</i> , <i>przesunięcie</i> , <i>bufor</i>)	Zapisuje <i>ilość</i> bajtów lub znaków ze zmiennej <i>bufor</i> do obiektu LOB, rozpoczynając w <i>przesunięciu</i> w LOB
WRITEAPPEND(<i>lob</i> , <i>ilość</i> , <i>bufor</i>)	Zapisuje na końcu obiektu LOB <i>ilość</i> znaków lub bajtów ze zmiennej <i>bufor</i>

W kolejnych podrozdziałach zostaną opisane wybrane metody z tej tabeli. Wszystkie metody z pakietu DBMS_LOB zostały opisane w podręczniku *Oracle Database PL/SQL Packages and Types Reference* opublikowanym przez Oracle Corporation.

APPEND()

Metoda APPEND() dodaje dane ze źródłowego obiektu LOB na koniec docelowego obiektu LOB. Istnieją dwie wersje metody APPEND():

```
DBMS_LOB.APPEND(
    lob_docelowy IN OUT NOCOPY BLOB,
    lob_źródłowy IN BLOB
);
DBMS_LOB.APPEND(
    lob_docelowy IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    lob_źródłowy IN CLOB/NCLOB CHARACTER SET lob_docelowy%CHARSET
);
```

gdzie:

- *lob_docelowy* jest docelowym obiektem LOB, do którego dołączane są dane,
- *lob_źródłowy* jest źródłowym obiektem LOB, z którego dane są odczytywane,
- CHARACTER SET ANY_CS oznacza, że dane w *lob_docelowy* mogą mieć dowolny zestaw znaków,
- CHARACTER SET *lob_docelowy*%CHARACTERSET jest zestawem znaków *lob_docelowy*.

W poniższej tabeli przedstawiono wyjątek zgłoszany przez metodę APPEND().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	<i>lob_docelowy</i> lub <i>lob_źródłowy</i> mają wartość NULL

CLOSE()

Metoda CLOSE() zamknuje otwarty obiekt LOB. Istnieją trzy wersje tej metody:

```
DBMS_LOB CLOSE(
    lob IN OUT NOCOPY BLOB
);
DBMS_LOB CLOSE(
    lob IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS
);
DBMS_LOB CLOSE(
    lob IN OUT NOCOPY BFILE
);
```

gdzie *lob* jest obiektem LOB, który ma zostać zamknięty.

COMPARE()

Metoda **COMPARE()** porównuje *ilość* bajtów lub znaków składowanych w dwóch obiektach LOB, rozpoznając w przesunięciach. Występują trzy wersje tej metody:

```
DBMS_LOB.COMPARE(
    lob1 IN BLOB,
    lob2 IN BLOB,
    ilość IN INTEGER := 4294967295,
    przesunięcie1 IN INTEGER := 1,
    przesunięcie2 IN INTEGER := 1
) RETURN INTEGER;

DBMS_LOB.COMPARE(
    lob1 IN CLOB/NCLOB CHARACTER SET ANY_CS,
    lob2 IN CLOB/NCLOB CHARACTER SET lob_1%CHARSET,
    ilość IN INTEGER := 4294967295,
    przesunięcie1 IN INTEGER := 1,
    przesunięcie2 IN INTEGER := 1
) RETURN INTEGER;

DBMS_LOB.COMPARE(
    lob1 IN BFILE,
    lob2 IN BFILE,
    ilość IN INTEGER,
    przesunięcie1 IN INTEGER := 1,
    przesunięcie2 IN INTEGER := 1
) RETURN INTEGER;
```

gdzie:

- *lob1* i *lob2* są porównywanyymi obiektami LOB,
- *ilosc* jest maksymalną liczbą znaków do odczytania z obiektu CLOB lub NCLOB lub maksymalną liczbą bajtów do odczytania z obiektu BLOB lub BFILE,
- *przesunięcie1* i *przesunięcie2* są przesunięciami w znakach lub bajtach w *lob1* i *lob2*, od których rozpoczyna się porównywanie (przesunięcia rozpoczynają się od 1).

Metoda **COMPARE()** zwraca:

- 0, jeżeli obiekty LOB są identyczne,
- 1, jeżeli obiekty LOB są różne,
- NULL, jeżeli:
 - *ilosc* < 1,
 - *ilosc* > LOBMAXSIZE (uwaga: LOBMAXSIZE jest maksymalnym rozmiarem obiektu LOB),
 - *przesunięcie1* lub *przesunięcie2* < 1,
 - *przesunięcie1* lub *przesunięcie2* > LOBMAXSIZE.

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę **COMPARE()**.

Wyjątek	Zgłaszanego, gdy
UNOPENED_FILE	Plik nie został otwarty
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Brak uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale brak jest wystarczających uprawnień dostępu do niego

COPY()

Metoda **COPY()** kopiuje znaki lub bajty ze źródłowego obiektu LOB do docelowego obiektu LOB, rozpoczynając w przesunięciach. Istnieją dwie wersje tej metody:

```
DBMS_LOB.COPY(
    lob_docelowy IN OUT NOCOPY BLOB,
    lob_źródłowy IN BLOB,
    ilość IN INTEGER,
    przesunięcie_doc IN INTEGER := 1,
    przesunięcie_źródł IN INTEGER := 1
);

DBMS_LOB.COPY(
    lob_docelowy IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    lob_źródłowy IN          CLOB/NCLOB CHARACTER SET lob_docelowy%CHARSET,
    ilość IN                INTEGER,
    przesunięcie_doc IN      INTEGER := 1,
    przesunięcie_źródł IN    INTEGER := 1
);
```

gdzie:

- *lob_docelowy* i *lob_źródłowy* są (odpowiednio) odczytywanym i zapisywanyem obiektem LOB,
- *ilosc* jest maksymalną liczbą znaków do odczytania z obiektu CLOB lub NCLOB lub maksymalną liczbą bajtów do odczytania z obiektu BLOB lub BFILE,
- *przesunięcie_doc* i *przesunięcie_źródł* są przesunięciami w znakach lub bajtach w *lob_docelowy* i *lob_źródłowy*, od których rozpoczęnie się kopiowanie (przesunięcia rozpoczynają się od 1).

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę **COPY()**.

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów ma wartość NULL
INVALID_ARGVAL	Któreś z poniższych: <ul style="list-style-type: none"> ■ <i>przesunięcie_źródł</i> < 1 ■ <i>przesunięcie_doc</i> < 1 ■ <i>przesunięcie_źródł</i> > LOBMAXSIZE ■ <i>przesunięcie_doc</i> > LOBMAXSIZE ■ <i>ilosc</i> < 1 ■ <i>ilosc</i> < LOBMAXSIZE

CREATETEMPORARY()

Metoda **CREATETEMPORARY()** tworzy tymczasowy obiekt LOB w domyślnej, tymczasowej przestrzeni tabel użytkownika. Istnieją dwie wersje tej metody:

```
DBMS_LOB.CREATETEMPORARY(
    lob   IN OUT NOCOPY BLOB,
    cache IN      BOOLEAN,
    czas  IN      PLS_INTEGER := DBMS_LOB.SESSION
);

DBMS_LOB.CREATETEMPORARY (
    lob   IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    cache IN      BOOLEAN,
    czas  IN      PLS_INTEGER := DBMS_LOB.SESSION
);
```

gdzie:

- *lob* jest tymczasowym obiektem LOB do utworzenia,
- *cache* określa, czy LOB powinien być wczytany do bufora pamięci podręcznej (`true`, jeżeli tak, `false`, jeżeli nie),
- *czas* (może mieć wartości `SESSION`, `TRANSACTION` lub `CALL`) określa, kiedy usunąć tymczasowy obiekt LOB — na końcu sesji, transakcji lub wywołania (domyślną wartością jest `SESSION`).

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę `CREATETEMPORARY()`.

Wyjątek	Zgłaszanego, gdy
<code>VALUE_ERROR</code>	Parametr <i>lob</i> ma wartość <code>NULL</code>

ERASE()

Metoda `ERASE()` usuwa określona liczbę znaków lub bajtów z obiektu LOB. Istnieją dwie wersje tej metody:

```
DBMS_LOB.ERASE(
    lob          IN OUT NOCOPY BLOB,
    liczba       IN OUT NOCOPY INTEGER,
    przesunięcie IN          INTEGER := 1
);

DBMS_LOB.ERASE(
    lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    liczba       IN OUT NOCOPY INTEGER,
    przesunięcie IN          INTEGER := 1
);
```

gdzie:

- *lob* jest obiektem LOB, którego dane będą usuwane,
- *liczba* jest maksymalną liczbą znaków do odczytania z obiektu `CLOB` lub `NCLOB` albo liczbą bajtów do odczytania z obiektu `BLOB`,
- *przesunięcie* jest przesunięciem w znakach lub bajtach, określającym rozpoczęcie usuwania (przesunięcie rozpoczyna się od 1).

W poniższej tabeli opisano wyjątki zgłaszane przez metodę `ERASE()`.

Wyjątek	Zgłaszanego, gdy
<code>VALUE_ERROR</code>	Którykolwiek z parametrów ma wartość <code>NULL</code>
<code>INVALID_ARGVAL</code>	Któreś z poniższych: <ul style="list-style-type: none"> ■ <i>liczba</i> < 1 ■ <i>liczba</i> > <code>LOBMAXSIZE</code> ■ <i>przesunięcie</i> < 1 ■ <i>przesunięcie</i> > <code>LOBMAXSIZE</code>

FILECLOSE()

Metoda `FILECLOSE()` zamknięta BFILE. Należy stosować nowszą procedurę, `CLOSE()`, ponieważ Oracle Corporation nie będzie już rozbudowywać `FILECLOSE()`. Opis tej metody został zamieszczony jedynie w celu ułatwienia zrozumienia starszych programów.

```
DBMS_LOB.FILECLOSE(
    bfile IN OUT NOCOPY BFILE
);
```

gdzie *bfile* jest obiektem `BFILE` do zamknięcia.

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę `FILECLOSE()`.

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Parametr <i>bfile</i> ma wartość NULL
UNOPENED_FILE	Plik nie został jeszcze otwarty
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Nie ma uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale nie ma wystarczających uprawnień dostępu do niego

FILECLOSEALL()

Metoda FILECLOSEALL() zamyka wszystkie obiekty BFILE.

DBMS_LOB.FILECLOSEALL;

W poniższej tabeli opisano wyjątki zgłoszane przez metodę FILECLOSEALL().

Wyjątek	Zgłaszanego, gdy
UNOPENED_FILE	W bieżącej sesji nie otwarto żadnych plików

FILEEXISTS()

Metoda FILEEXISTS() sprawdza, czy plik istnieje.

```
DBMS_LOB.FILEEXISTS(
  bfile IN BFILE
) RETURN INTEGER;
```

gdzie *bfile* jest obiektem BFILE wskazującym na plik zewnętrzny.

Metoda FILEEXISTS() zwraca:

- 0, jeżeli plik nie istnieje,
- 1, jeżeli plik istnieje.

W poniższej tabeli opisano wyjątki zgłoszane przez metodę FILEEXISTS().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Parametr <i>bfile</i> ma wartość NULL
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Nie ma uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy

FILEGETNAME()

Metoda FILEGETNAME() zwraca katalog oraz nazwę pliku z obiektu BFILE.

```
DBMS_LOB.FILEGETNAME(
  bfile      IN BFILE,
  katalog    OUT VARCHAR2,
  nazwa_pliku OUT VARCHAR2
);
```

gdzie:

- *bfile* jest wskaźnikiem do pliku,
- *katalog* jest katalogiem, w którym znajduje się plik,
- *nazwa_pliku* jest nazwą pliku.

W poniższej tabeli opisano wyjątki zgłoszane przez metodę FILEGETNAME().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych jest nieprawidłowy
INVALID_ARGVAL	Parametr <i>katalog</i> lub <i>nazwa_pliku</i> ma wartość NULL

FILEISOPEN()

Metoda FILEISOPEN() sprawdza, czy plik jest otwarty. W nowych programach należy używać w tym celu nowszej procedury, ISOPEN(), ponieważ metoda FILEISOPEN() nie będzie już rozwijana przez Oracle Corporation. Jej opis został zamieszczony jedynie w celu ułatwienia zrozumienia starszych programów.

```
DBMS_LOB.FILEISOPEN(
  bfile IN BFILE
) RETURN INTEGER;
```

gdzie *bfile* jest wskaźnikiem do pliku.

Metoda FILEISOPEN() zwraca:

- 0, jeżeli plik nie jest otwarty,
- 1, jeżeli plik jest otwarty.

W poniższej tabeli opisano wyjątki zgłoszane przez metodę FILEISOPEN().

Wyjątek	Zgłaszanego, gdy
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Nie ma uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale nie ma wystarczających uprawnień dostępu do niego

FILEOPEN()

Metoda FILEOPEN() otwiera plik. W nowych programach należy używać w tym celu nowszej procedury OPEN(), ponieważ metoda FILEOPEN() nie będzie już rozwijana przez Oracle Corporation. Jej opis został zamieszczony jedynie w celu ułatwienia zrozumienia starszych programów.

```
DBMS_LOB.FILEOPEN(
  bfile IN OUT NOCOPY BFILE,
  tryb IN          BINARY_INTEGER := DBMS_LOB.FILE_READONLY
);
```

gdzie:

- *bfile* jest wskaźnikiem do pliku,
- *tryb* określa tryb otwarcia. Jedynym dostępnym trybem jest DBMS_LOB.FILE_READONLY, co oznacza, że plik może być tylko odczytywany.

W poniższej tabeli opisano wyjątki zgłoszane przez metodę FILEOPEN().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
INVALID_ARGVAL	Parametr <i>tryb</i> ma wartość inną niż DBMS_LOB.FILE_READONLY
OPEN_TOOMANY	Nastąpiła próba otwarcia większej liczby plików niż SESSION_MAX_OPEN_FILES. SESSION_MAX_OPEN_FILES jest parametrem inicjalizującym bazę danych, ustawianym przez administratora
NOEXIST_DIRECTORY	Katalog nie istnieje
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale nie ma wystarczających uprawnień dostępu do niego

FREETEMPORARY()

Metoda FREETEMPORARY() zwalnia tymczasowy obiekt LOB z domyślnej tymczasowej przestrzeni tabel użytkownika. Istnieją dwie wersje tej metody:

```
DBMS_LOB.FREETEMPORARY (
    lob IN OUT NOCOPY BLOB
);
DBMS_LOB.FREETEMPORARY (
    lob IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS
);
```

gdzie *lob* jest zwalnianym obiektem LOB.

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę FREETEMPORARY().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość NULL

GETCHUNKSIZE()

Metoda GETCHUNKSIZE() zwraca rozmiar fragmentu używanego przy odczytywaniu i zapisywaniu danych LOB (fragment jest jednostką danych). Istnieją dwie wersje tej metody:

```
DBMS_LOB.GETCHUNKSIZE(
    lob IN BLOB
) RETURN INTEGER;
DBMS_LOB.GETCHUNKSIZE(
    lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
```

gdzie *lob* jest obiektem LOB, dla którego pobierany jest rozmiar fragmentu.

Metoda GETCHUNKSIZE() zwraca:

- rozmiar fragmentu w bajtach w przypadku obiektu BLOB,
- rozmiar fragmentu w znakach w przypadku obiektu CLOB lub NCLOB.

W poniższej tabeli opisano wyjątek zgłaszany przez metodę GETCHUNKSIZE().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Parametr <i>lob</i> ma wartość NULL

GETLENGTH()

Metoda GETLENGTH() zwraca długość danych obiektu LOB. Istnieją trzy wersje tej metody:

```
DBMS_LOB.GETLENGTH(
    lob IN BLOB
) RETURN INTEGER;
DBMS_LOB.GETLENGTH(
    lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
DBMS_LOB.GETLENGTH(
    bfile IN BFILE
) RETURN INTEGER;
```

gdzie:

- *lob* jest obiektem BLOB, CLOB lub NCLOB, którego długość jest obliczana,
- *bfile* jest obiektem BFILE, którego długość jest obliczana.

Metoda `GETLENGTH()` zwraca:

- długość w bajtach w przypadku obiektu **BLOB** lub **BFILE**,
- długość w znakach w przypadku obiektu **CLOB** lub **NCLOB**.

W poniższej tabeli opisano wyjątek zgłoszany przez metodę `GETLENGTH()`.

Wyjątek	Zgłoszony, gdy
<code>VALUE_ERROR</code>	Parametr <i>lob</i> lub <i>bfile</i> ma wartość NULL

GET_STORAGE_LIMIT()

Metoda `GET_STORAGE_LIMIT()` zwraca maksymalny dopuszczalny rozmiar obiektu LOB.

```
DBMS_LOB.GET_STORAGE_LIMIT(
  lob IN CLOB CHARACTER SET ANY_CS
) RETURN INTEGER;

DBMS_LOB.GET_STORAGE_LIMIT(
  lob IN BLOB
) RETURN INTEGER;
```

gdzie *lob* jest obiektem, którego limit wielkości ma być pobrany.

INSTR()

Metoda `INSTR()` zwraca początkową pozycję znaków stanowiących *n*-te wystąpienie wzorca w danych LOB (przeszukiwanie rozpoczyna się od przesunięcia). Istnieją trzy wersje tej metody:

```
DBMS_LOB.INSTR(
  lob          IN BLOB,
  wzorzec     IN RAW,
  przesunięcie IN INTEGER := 1,
  n            IN INTEGER := 1
) RETURN INTEGER;

DBMS_LOB.INSTR(
  lob          IN CLOB/NCLOB CHARACTER SET ANY_CS,
  wzorzec     IN VARCHAR2 CHARACTER SET lob%CHARSET,
  przesunięcie IN INTEGER := 1,
  n            IN INTEGER := 1
) RETURN INTEGER;

DBMS_LOB.INSTR(
  bfile        IN BFILE,
  wzorzec     IN RAW,
  przesunięcie IN INTEGER := 1,
  n            IN INTEGER := 1
) RETURN INTEGER;
```

gdzie:

- *lob* jest odczytywanym obiektem **BLOB**, **CLOB** lub **NCLOB**.
- *bfile* jest odczytywanym obiektem **BFILE**.
- *wzorzec* jest wzorcem wyszukiwanym w danych LOB. W przypadku obiektów **BLOB** i **BFILE** wzorzec jest grupą bajtów **RAW**, w przypadku obiektów **CLOB** jest natomiast napisem **VARCHAR2**. Maksymalny rozmiar wzorca wynosi 16 383 bajtów.
- *przesunięcie* jest przesunięciem, od którego rozpocznie się odczyt danych z obiektu LOB (przesunięcie rozpoczyna się od 1).
- *n* jest wyszukiwanym wystąpieniem wzorca.

Metoda **INSTR()** zwraca:

- przesunięcie rozpoczęcia wzorca (jeżeli zostanie odnaleziony),
- 0, jeżeli wzorzec nie zostanie odnaleziony,
- NULL, jeżeli:
 - któryś z parametrów IN ma wartość NULL lub jest nieprawidłowy,
 - przesunięcie < 1 lub przesunięcie > LOBMAXSIZE,
 - N < 1 lub n > LOBMAXSIZE.

W poniższej tabeli opisano wyjątki zgłaszane przez metodę **INSTR()**.

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
UNOPENED_FILE	Plik nie został otwarty
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Nie ma uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale nie ma wystarczających uprawnień dostępu do niego

ISOPEN()

Metoda **ISOPEN()** sprawdza, czy obiekt LOB jest otwarty. Istnieją trzy wersje tej metody:

```
DBMS_LOB.ISOPEN(
  lob IN BLOB
) RETURN INTEGER;

DBMS_LOB.ISOPEN(
  lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;

DBMS_LOB.ISOPEN(
  bfile IN BFILE
) RETURN INTEGER;
```

gdzie:

- *lob* jest sprawdzanym obiektem BLOB, CLOB lub NCLOB,
- *bfile* jest sprawdzanym obiektem BFILE.

Metoda **ISOPEN()** zwraca:

- 0, jeżeli obiekt LOB nie jest otwarty,
- 1, jeżeli obiekt LOB jest otwarty.

W poniższej tabeli opisano wyjątek zgłaszany przez metodę **ISOPEN()**.

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Parametr <i>lob</i> lub <i>bfile</i> ma wartość NULL lub jest nieprawidłowy

ISTEMPORARY()

Metoda **ISTEMPORARY()** sprawdza, czy obiekt LOB jest obiektem tymczasowym. Istnieją dwie wersje tej metody:

```
DBMS_LOB.ISTEMPORARY(
  lob IN BLOB
) RETURN INTEGER;
```

```
DBMS_LOB.ISTEMPORARY (
  lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
```

gdzie *lob* jest sprawdzanym obiektem LOB.

Metoda ISTEMPORARY() zwraca:

- 0, jeżeli obiekt LOB nie jest tymczasowy,
- 1, jeżeli obiekt LOB jest tymczasowy.

W poniższej tabeli przedstawiono wyjątek zgłoszany przez metodę ISTEMPORARY().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Parametr <i>lob</i> ma wartość NULL lub jest nieprawidłowy

LOADFROMFILE()

Metoda LOADFROMFILE() wczytuje do obiektu CLOB, NCLOB lub BLOB określoną liczbę znaków lub bajtów pobranych za pośrednictwem BFILE, rozpoczynając od przesunięć. W nowych programach należy używać w tym celu bardziej wydajnych procedur, LOADCLOBFROMFILE() i LOADBLOBFROMFILE(). Opis metody LOADFROMFILE() został zamieszczony jedynie w celu ułatwienia zrozumienia starszych programów.

Istnieją dwie wersje metody LOADFROMFILE():

```
DBMS_LOB.LOADFROMFILE(
  lob_doc IN OUT NOCOPY BLOB,
  bfile_źródł IN BFILE,
  liczba IN INTEGER,
  przesunięcie_doc IN INTEGER := 1,
  przesunięcie_źródł IN INTEGER := 1
);
DBMS_LOB.LOADFROMFILE(
  lob_doc      IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  bfile_źródł   IN          BFILE,
  liczba        IN          INTEGER,
  przesunięcie_doc IN          INTEGER := 1,
  przesunięcie_źródł IN          INTEGER := 1
);
```

gdzie:

- *lob_doc* jest obiektem LOB, do którego będą zapisywane dane,
- *bfile_źródł* jest wskaźnikiem do pliku, z którego będą odczytywane dane,
- *liczba* jest maksymalną liczbą bajtów lub znaków do odczytania z *bfile_źródł*,
- *przesunięcie_doc* jest przesunięciem (w bajtach lub w znakach) w *lob_doc*, od którego rozpoczęnie się zapis (rozpoczęcie rozpoczęna się od 1),
- *przesunięcie_źródł* jest przesunięciem w bajtach w *bfile_źródł*, od którego rozpoczęnie się odczyt danych (przesunięcie rozpoczęna się od 1).

W poniższej tabeli przedstawiono wyjątki zgłoszane przez metodę LOADFROMFILE().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
INVALID_ARGVAL	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <i>przesunięcie_źródł</i> < 1 ■ <i>przesunięcie_doc</i> < 1 ■ <i>przesunięcie_źródł</i> > LOBMAXSIZE ■ <i>przesunięcie_doc</i> > LOBMAXSIZE ■ <i>liczba</i> < 1 ■ <i>liczba</i> > LOBMAXSIZE

LOADBLOBFROMFILE()

Metoda `LOADBLOBFROMFILE()` wczytuje do obiektu `BLOB` dane odczytane za pośrednictwem `BFILE`. Oferuje ona większą wydajność niż metoda `LOADFROMFILE()` w przypadku pracy z obiektami `BLOB`.

```
DBMS_LOB.LOADBLOBFROMFILE(
  blob_doc      IN OUT NOCOPY BLOB,
  bfile_źródł   IN BFILE,
  liczba        IN INTEGER,
  przesunięcie_doc  IN OUT INTEGER := 1,
  przesunięcie_źródł IN OUT INTEGER := 1
);
```

gdzie:

- `blob_doc` jest obiektem `BLOB`, do którego będą zapisywane dane,
- `bfile_źródł` jest wskaźnikiem do pliku, z którego będą odczytywane dane,
- `liczba` jest maksymalną liczbą bajtów do odczytania z `bfile_źródł`,
- `przesunięcie_doc` jest wyrażonym w bajtach przesunięciem w `blob_doc`, od którego rozpocznie się zapis danych (przesunięcie rozpoczyna się od 1),
- `przesunięcie_źródł` jest wyrażonym w bajtach przesunięciem w `bfile_źródł`, od którego rozpocznie się odczyt danych (przesunięcie rozpoczyna się od 1).

W poniższej tabeli opisano wyjątki zgłoszane przez metodę `LOADBLOBFROMFILE()`.

Wyjątek	Zgłaszany, gdy
<code>VALUE_ERROR</code>	Którykolwiek z parametrów wejściowych ma wartość <code>NULL</code> lub jest nieprawidłowy
<code>INVALID_ARGVAL</code>	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <code>przesunięcie_źródł < 1</code> ■ <code>przesunięcie_doc < 1</code> ■ <code>przesunięcie_źródł > LOBMAXSIZE</code> ■ <code>przesunięcie_doc > LOBMAXSIZE</code> ■ <code>liczba < 1</code> ■ <code>liczba > LOBMAXSIZE</code>

LOADCLOBFROMFILE()

Metoda `LOADCLOBFROMFILE()` wczytuje do obiektu `CLOB` (i `NCLOB`) dane odczytane za pośrednictwem `BFILE`. Oferuje ona większą wydajność niż metoda `LOADFROMFILE()` w przypadku pracy z obiektami `CLOB` i `NCLOB`. Ponadto automatycznie konwertuje dane binarne na znakowe.

```
DBMS_LOB.LOADCLOBFROMFILE(
  blob_doc      IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  bfile_źródł   IN          BFILE,
  liczba        IN          INTEGER,
  przesunięcie_doc  IN OUT    INTEGER,
  przesunięcie_źródł IN OUT    INTEGER,
  csid_źródł    IN          NUMBER,
  kontekst_językowy IN OUT    INTEGER,
  ostrzeżenie    OUT         INTEGER
);
```

gdzie:

- `blob_doc` jest obiektem `CLOB` (`NCLOB`), w którym będą zapisywane dane,
- `bfile_źródł` jest wskaźnikiem do pliku, z którego będą odczytywane dane,
- `liczba` jest maksymalną liczbą bajtów do odczytania z `bfile_źródł`,
- `przesunięcie_doc` jest wyrażonym w bajtach przesunięciem w `blob_doc`, od którego rozpocznie się zapis danych (przesunięcie rozpoczyna się od 1),

- *przesunięcie_źródł* jest wyrażonym w bajtach przesunięciem w *bfile_źródł*, od którego rozpocznie się odczyt danych (przesunięcie rozpoczyna się od 1),
- *csid_źródł* jest zestawem znaków w *bfile_źródł* (zwykle stosowana jest opcja `DBMS_LOB.DEFAULT_CSID`, czyli domyślny zestaw znaków w bazie danych),
- *kontekst_językowy* jest kontekstem językowym używanym przy wczytywaniu danych (zwykle stosowana jest opcja `DBMS_LOB.DEFAULT_LANG_CTX`, czyli domyślny kontekst językowy w bazie danych),
- *ostrzeżenie* jest komunikatem wyświetlany, gdy wystąpią błędy przy ładowaniu. Często następuje sytuacja, w której znak z *bfile_źródł* nie może zostać przekonwertowany na znak w *lob_doc* (w takim przypadku *ostrzeżenie* ma wartość `DBMS_LOB.WARN_INCONVERTIBLE_CHAR`).



Szczegółowe informacje na temat zestawów znaków, kontekstów oraz konwersji znaków z jednego języka na inny można znaleźć w podręczniku *Oracle Database Globalization Support Guide* opublikowanym przez Oracle Corporation.

W poniższej tabeli przedstawiono wyjątki zgłasiane przez metodę `LOADCLOBFROMFILE()`.

Wyjątek	Zgłaszanego, gdy
<code>VALUE_ERROR</code>	Którykolwiek z parametrów wejściowych ma wartość <code>NULL</code> lub jest nieprawidłowy
<code>INVALID_ARGVAL</code>	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <i>przesunięcie_źródł</i> < 1 ■ <i>przesunięcie_doc</i> < 1 ■ <i>przesunięcie_źródł</i> > <code>LOBMAXSIZE</code> ■ <i>przesunięcie_doc</i> > <code>LOBMAXSIZE</code> ■ <i>liczba</i> < 1 ■ <i>liczba</i> > <code>LOBMAXSIZE</code>

OPEN()

Metoda `OPEN()` otwiera obiekt LOB. Występują trzy wersje tej metody:

```
DBMS_LOB.OPEN(
  lob  IN OUT NOCOPY BLOB,
  tryb IN BINARY_INTEGER
);
DBMS_LOB.OPEN(
  lob  IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  tryb IN BINARY_INTEGER
);
DBMS_LOB.OPEN(
  bfile IN OUT NOCOPY BFILE,
  tryb IN BINARY_INTEGER := DBMS_LOB.FILE_READONLY
);
```

gdzie:

- *lob* jest obiektem LOB do otwarcia,
- *bfile* jest wskaźnikiem do otwieranego pliku,
- *tryb* jest trybem otwarcia pliku. Domyślnie jest ustawiany tryb `DBMS_LOB.FILE_READONLY`, w którym obiekt LOB może być jedynie odczytywany. Ustawienie `DBMS_LOB.FILE_READWRITE` dopuszcza zarówno odczyt, jak i zapis do obiektu LOB.

W poniższej tabeli opisano wyjątek zgłaszanego przez metodę `OPEN()`.

Wyjątek	Zgłaszanego, gdy
<code>VALUE_ERROR</code>	Którykolwiek z parametrów wejściowych ma wartość <code>NULL</code> lub jest nieprawidłowy

READ()

Metoda READ() odczytuje do bufora dane z obiektu LOB. Występują trzy wersje tej metody:

```
DBMS_LOB.READ(
    lob          IN      BLOB,
    liczba      IN OUT NOCOPY BINARY_INTEGER,
    przesunięcie IN      INTEGER,
    bufor       OUT      RAW
);

DBMS_LOB.READ(
    lob          IN      CLOB/NCLOB CHARACTER SET ANY_CS,
    liczba      IN OUT NOCOPY BINARY_INTEGER,
    przesunięcie IN      INTEGER,
    bufor       OUT      VARCHAR2 CHARACTER SET lob%CHARSET
);

DBMS_LOB.READ(
    bfile        IN      BFILE,
    liczba      IN OUT NOCOPY BINARY_INTEGER,
    przesunięcie IN      INTEGER,
    bufor       OUT      RAW
);
```

gdzie:

- *lob* jest czytanym obiektem CLOB, NCLOB lub BLOB,
- *bfile* jest czytanym obiektem BFILE,
- *liczba* jest maksymalną liczbą znaków do odczytania z obiektu CLOB (NCLOB) lub maksymalną liczbą bajtów do odczytania z obiektu BLOB (BFILE),
- *przesunięcie* jest przesunięciem, od którego rozpocznie się odczyt (przesunięcia rozpoczynają się od 1),
- *bufor* jest zmienną, w której będą składowane dane odczytane z obiektu LOB.

W poniższej tabeli opisano wyjątki zgłoszane przez metodę READ().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którekolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
INVALID_ARGVAL	Którekolwiek z poniższych:
	<ul style="list-style-type: none"> ■ <i>liczba</i> < 1 ■ <i>liczba</i> > MAXBUFSIZE ■ <i>liczba</i> > pojemności bufora wyrażonej w znakach lub bajtach ■ <i>przesunięcie</i> < 1 ■ <i>przesunięcie</i> > LOBMAXSIZE
NO_DATA_FOUND	Osiągnięto koniec LOB, więc nie można odczytać dalszych bajtów lub znaków z LOB

SUBSTR()

Metoda SUBSTR() zwraca określoną liczbę znaków lub bajtów z obiektu LOB, rozpoczynając w przesunięciu. Istnieją trzy wersje tej metody:

```
DBMS_LOB.SUBSTR(
    lob          IN BLOB,
    liczba      IN INTEGER := 32767,
    przesunięcie IN INTEGER := 1
) RETURN RAW;

DBMS_LOB.SUBSTR (
    lob          IN CLOB/NCLOB CHARACTER SET ANY_CS,
```

```

    liczba      IN INTEGER := 32767,
    przesunięcie IN INTEGER := 1
) RETURN VARCHAR2 CHARACTER SET lob%CHARSET;

DBMS_LOB.SUBSTR (
    bfile       IN BFILE,
    liczba      IN INTEGER := 32767,
    przesunięcie IN INTEGER := 1
) RETURN RAW;

```

gdzie:

- *lob* jest odczytywanym obiektem BLOB, CLOB lub NCLOB,
- *bfile* jest wskaźnikiem do czytanego pliku,
- *liczba* jest maksymalną liczbą znaków do odczytania z obiektu CLOB (NCLOB) lub maksymalną liczbą bajtów do odczytania z obiektu BLOB (BFILE),
- *przesunięcie* jest przesunięciem, od którego rozpocznie się odczyt (przesunięcia rozpoczynają się od 1).

Metoda SUBSTR() zwraca:

- dane typu RAW, jeżeli czytany jest obiekt BLOB (BFILE),
- dane typu VARCHAR2, jeżeli czytany jest obiekt typu CLOB (NCLOB),
- NULL, jeżeli:
 - *liczba* < 1,
 - *liczba* > 32767,
 - *przesunięcie* < 1,
 - *przesunięcie* > LOBMAXSIZE.

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę SUBSTR().

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
UNOPENED_FILE	Obiekt BFILE nie został otwarty
NOEXIST_DIRECTORY	Katalog nie istnieje
NOPRIV_DIRECTORY	Nie ma uprawnień dostępu do katalogu
INVALID_DIRECTORY	Katalog jest nieprawidłowy
INVALID_OPERATION	Plik istnieje, ale nie ma wystarczających uprawnień dostępu do niego

TRIM()

Metoda TRIM() przycinana dane obiektu LOB do określonej długości. Istnieją dwie wersje tej metody:

```

DBMS_LOB.TRIM(
    lob        IN OUT NOCOPY BLOB,
    nowa_dł   IN INTEGER
);

DBMS_LOB.TRIM(
    lob        IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    nowa_dł   IN INTEGER
);

```

gdzie:

- *lob* jest przycinanym obiektem typu BLOB, CLOB lub NCLOB,
- *nowa_dł* jest nową długością (w bajtach w przypadku obiektu BLOB, w znakach w przypadku obiektów CLOB i NCLOB).

W poniższej tabeli opisano wyjątki zgłaszane przez metodę TRIM().

Wyjątek	Zgłaszany, gdy
VALUE_ERROR	Parametr <i>lob</i> ma wartość NULL
INVALID_ARGVAL	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <i>nowa_dł</i> < 0 ■ <i>nowa_dł</i> > LOBMAXSIZE

WRITE()

Metoda WRITE() zapisuje dane z bufora do obiektu LOB. Istnieją dwie wersje tej metody:

```
DBMS_LOB.WRITE(
  lob      IN OUT NOCOPY BLOB,
  liczba   IN          INTEGER,
  przesunięcie IN          INTEGER,
  bufor    IN          RAW
);
DBMS_LOB.WRITE(
  lob      IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  liczba   IN          INTEGER,
  przesunięcie IN          INTEGER,
  bufor    IN          VARCHAR2 CHARACTER SET lob%CHARSET
);
```

gdzie:

- *lob* jest zapisywanym obiektem LOB,
- *liczba* jest maksymalną liczbą znaków do zapisania do obiektu CLOB (NCLOB) lub maksymalną liczbą bajtów do zapisania do obiektu BLOB,
- *przesunięcie* jest przesunięciem, od którego rozpocznie się zapis (przesunięcia rozpoczynają się od 1),
- *bufor* jest zmienną zawierającą dane do zapisania w obiekcie LOB.

W poniższej tabeli przedstawiono wyjątki zgłaszane przez metodę WRITE().

Wyjątek	Zgłaszany, gdy
VALUE_ERROR	Którekolwiek z parametrów wejściowych ma wartość NULL lub jest nieprawidłowy
INVALID_ARGVAL	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <i>liczba</i> < 1 ■ <i>liczba</i> > MAXBUFSIZE ■ <i>przesunięcie</i> < 1 ■ <i>przesunięcie</i> > LOBMAXSIZE

WRITEAPPEND()

Metoda WRITEAPPEND() zapisuje określoną liczbę bajtów lub znaków z bufora na końcu obiektu LOB. Istnieją dwie wersje tej metody:

```
DBMS_LOB.WRITEAPPEND(
  lob IN OUT NOCOPY BLOB,
  liczba IN BINARY_INTEGER,
  bufor IN RAW
);
DBMS_LOB.WRITEAPPEND(
  lob  IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  liczba IN BINARY_INTEGER,
  bufor IN VARCHAR2 CHARACTER SET lob%CHARSET
);
```

gdzie:

- *lob* jest zapisywanym obiektem LOB,
- *liczba* jest maksymalną liczbą znaków do zapisania do obiektu CLOB (NCLOB) lub maksymalną liczbą bajtów do zapisania do obiektu BLOB,
- *bufor* jest zmienną zawierającą dane do zapisania w obiekcie LOB.

W poniższej tabeli przedstawiono wyjątki zgłoszane przez metodę `WRITEAPPEND()`.

Wyjątek	Zgłaszanego, gdy
VALUE_ERROR	Którykolwiek z parametrów wejściowych ma wartość <code>NULL</code> lub jest nieprawidłowy
INVALID_ARGVAL	Którekolwiek z poniższych: <ul style="list-style-type: none"> ■ <i>liczba</i> < 1 ■ <i>liczba</i> > MAXBUFSIZE

Przykładowe procedury PL/SQL

W tym podrozdziale zostaną przedstawione procedury PL/SQL wykorzystujące różne metody opisane w poprzednim podrozdziale. Te procedury zostały utworzone przez skrypt `lob_schema.sql`.

Pobieranie lokalizatora LOB

Poniższa procedura `get_clob_locator()` pobiera lokalizator obiektu LOB z tabeli `clob_content`. Procedura ta wykonuje następujące zadania:

- Przyjmuje parametr IN OUT o nazwie `p_clob` i typie CLOB. Wewnątrz procedury ten parametr jest ustawiany na lokalizator obiektu LOB. Ponieważ `p_clob` jest parametrem o trybie IN OUT, wartość jest zwracana przez procedurę.
- Przyjmuje parametr IN o nazwie `p_id` i typie INTEGER, określającym id wiersza, który ma zostać pobrany z tabeli `clob_content`.
- Wybiera kolumnę `clob_column` z tabeli `clob_content` do `p_clob`. W ten sposób do parametru `p_clob` jest zapisywany lokalizator LOB z `clob_column`.

```
CREATE PROCEDURE get_clob_locator(
  p_clob IN OUT CLOB,
  p_id IN INTEGER
) AS
BEGIN
  -- pobiera lokalizator LOB i zapisuje go w p_clob
  SELECT clob_column
  INTO p_clob
  FROM clob_content
  WHERE id = p_id;
END get_clob_locator;
/
```

Poniższa procedura `get_blob_locator()` wykonuje takie same czynności co poprzednia, z tą różnicą, że pobiera lokalizator obiektu BLOB z tabeli `blob_content`:

```
CREATE PROCEDURE get_blob_locator(
  p_blob IN OUT BLOB,
  p_id IN INTEGER
) AS
BEGIN
  -- pobiera lokalizator LOB i zapisuje go w p_blob
  SELECT blob_column
  INTO p_blob
  FROM blob_content
  WHERE id = p_id;
```

```

    WHERE id = p_id;
END get_blob_locator;
/

```

Te dwie procedury zostaną użyte w kodzie prezentowanym w dalszych podrozdziałach.

Odczyt danych z obiektów CLOB i BLOB

Poniższa procedura `read_clob_example()` odczytuje dane z obiektu CLOB i wyświetla je na ekranie. Wykonuje następujące czynności:

- pobiera lokalizator, wywołując procedurę `get_clob_locator()`, i zapisuje go w zmiennej `v_clob`,
- korzystając z metody `READ()`, odczytuje zawartość `v_clob` do zmiennej `v_char_buffer` typu `VARCHAR2`,
- wyświetla zawartość `v_char_buffer` na ekranie.

```

CREATE PROCEDURE read_clob_example(
    p_id IN INTEGER
) AS
    v_clob CLOB;
    v_offset INTEGER := 1;
    v_amount INTEGER := 50;
    v_char_buffer VARCHAR2(50);
BEGIN
    -- pobiera lokalizator LOB i zapisuje go w v_clob
    get_clob_locator(v_clob, p_id);

    -- odczytuje v_amount znaków z v_clob do v_char_buffer,
    -- rozpoczynając w pozycji określonej przez v_offset
    DBMS_LOB.READ(v_clob, v_amount, v_offset, v_char_buffer);

    -- wyświetla zawartość v_char_buffer
    DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);
    DBMS_OUTPUT.PUT_LINE('v_amount = ' || v_amount);
END read_clob_example;
/

```

W poniższym przykładzie włączono wypisywanie z serwera i wywołano procedurę `read_clob_example()`:

```

SET SERVEROUTPUT ON
CALL read_clob_example(1);
v_char_buffer = Wije się w ciasnym kółku
v_amount = 24

```

Poniższa procedura `read_blob_example()` odczytuje dane z obiektu BLOB. Wykonuje następujące czynności:

- pobiera lokalizator, wywołując procedurę `get_blob_locator()`, i zapisuje go w zmiennej `v_blob`,
- korzystając z metody `READ()`, odczytuje zawartość `v_blob` do zmiennej `v_binary_buffer` typu `RAW`,
- wyświetla zawartość `v_binary_buffer` na ekranie.

```

CREATE PROCEDURE read_blob_example(
    p_id IN INTEGER
) AS
    v_blob BLOB;
    v_offset INTEGER := 1;
    v_amount INTEGER := 25;
    v_binary_buffer RAW(25);
BEGIN
    -- pobiera lokalizator obiektu LOB i zapisuje go w v_blob
    get_blob_locator(v_blob, p_id);

    -- odczytuje v_amount bajtów z v_blob do v_binary_buffer,
    -- rozpoczynając w pozycji określonej przez v_offset
    DBMS_LOB.READ(v_blob, v_amount, v_offset, v_binary_buffer);

```

```
-- wyświetla zawartość v_binary_buffer
DBMS_OUTPUT.PUT_LINE('v_binary_buffer = ' || v_binary_buffer);
DBMS_OUTPUT.PUT_LINE('v_amount = ' || v_amount);
END read_blob_example;
/
```

W poniższym przykładzie wywołano procedurę `read_blob_example()`:

```
CALL read_blob_example(1);
v_binary_buffer = 100111010101011111
v_amount = 9
```

Zapisywanie do obiektu CLOB

Poniższa procedura `write_example()` zapisuje za pomocą metody `WRITE()` napis z `v_char_buffer` do `v_clob`. Należy zauważyć, że w instrukcji `SELECT` zastosowano klauzulę `FOR UPDATE`, co było konieczne, ponieważ zapis do obiektu `CLOB` będzie przeprowadzany za pomocą metody `WRITE()`:

```
CREATE PROCEDURE write_example(
    p_id IN INTEGER
) AS
    v_clob CLOB;
    v_offset INTEGER := 9;
    v_amount INTEGER := 6;
    v_char_buffer VARCHAR2(10) := 'ciągłe';
BEGIN
    -- pobiera lokalizator LOB do v_clob w celu modyfikacji (w celu modyfikacji,
    -- ponieważ do LOB będzie później przeprowadzany zapis za pomocą WRITE()
    SELECT clob_column
    INTO v_clob
    FROM clob_content
    WHERE id = p_id
    FOR UPDATE;

    -- czyta i wyświetla zawartość CLOB
    read_clob_example(p_id);

    -- zapisuje znaki z v_char_buffer do v_clob, rozpoczynając
    -- w pozycji określonej przez przesunięcie v_offset
    -- zapisuje v_amount znaków
    DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

    -- czyta i wyświetla zawartość CLOB,
    -- a następnie wycofuje zapis
    read_clob_example(p_id);
    ROLLBACK;
END write_example;
/
```

Poniżej przedstawiono wynik wywołania procedury `write_example()`:

```
CALL write_example(1);
v_char_buffer = Wije się w ciasnym kółku
v_amount = 24
v_char_buffer = Wije się ciąglesnym kółku
v_amount = 24
```

Dołączanie danych do obiektu CLOB

W poniższej procedurze `append_example()` użyto metody `APPEND()` do skopiowania danych z `v_src_clob` na koniec `v_dest_clob`:

```
CREATE PROCEDURE append_example AS
    v_src_clob CLOB;
    v_dest_clob CLOB;
```

```

BEGIN
  -- pobiera do v_src_clob lokalizator LOB obiektu
  -- CLOB znajdującego się w wierszu 2 tabeli clob_content
  get_clob_locator(v_src_clob, 2);

  -- pobiera w celu modyfikacji do v_dest_clob lokalizator LOB obiektu CLOB
  -- znajdującego się w wierszu 1 tabeli clob_content
  -- (w celu modyfikacji, ponieważ później za pomocą metody APPEND() zostanie
  -- dołączony obiekt CLOB
  SELECT clob_column
  INTO v_dest_clob
  FROM clob_content
  WHERE id = 1
  FOR UPDATE;

  -- czyta i wyświetla zawartość CLOB nr 1
  read_clob_example(1);

  -- używa APPEND() do skopiowania zawartości v_src_clob do v_dest_clob
  DBMS_LOB.APPEND(v_dest_clob, v_src_clob);

  -- czyta i wyświetla zawartość CLOB nr 1,
  -- a następnie wycofuje zmianę
  read_clob_example(1);
  ROLLBACK;
END append_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `append_example()`:

```

CALL append_example();
v_char_buffer = Wije się w ciasnym kółku
v_amount = 24
v_char_buffer = Wije się w ciasnym kółku od dnia do dnia
v_amount = 41

```

Porównywanie danych w dwóch obiektach CLOB

Poniższa procedura `compare_example()` porównuje za pomocą metody `COMPARE()` dane w obiektach `v_clob1` i `v_clob2`:

```

CREATE PROCEDURE compare_example AS
  v_clob1 CLOB;
  v_clob2 CLOB;
  v_return INTEGER;
BEGIN
  -- pobiera lokalizatory LOB
  get_clob_locator(v_clob1, 1);
  get_clob_locator(v_clob2, 2);

  -- porównuje v_clob1 z v_clob2 (COMPARE() zwraca 1,
  -- ponieważ zawartość v_clob1 z v_clob2 jest różna)
  DBMS_OUTPUT.PUT_LINE('Porównywanie v_clob1 i v_clob2');
  v_return := DBMS_LOB.COMPARE(v_clob1, v_clob2);
  DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

  -- porównuje v_clob1 z v_clob1 (COMPARE() zwraca 0,
  -- ponieważ zawartość obiektów jest identyczna)
  DBMS_OUTPUT.PUT_LINE('Porównywanie v_clob1 z v_clob1');
  v_return := DBMS_LOB.COMPARE(v_clob1, v_clob1);
  DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);
END compare_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `compare_example()`:

```
CALL compare_example();
Porównywanie v_clob1 i v_clob2
v_return = 1
Porównywanie v_clob1 z v_clob1
v_return = 0
```

Należy zauważać, że przy porównywaniu v_clob1 z v_clob2 zmienna v_return ma wartość 1, co oznacza, że dane w obiektach LOB są różne. Zmienna v_return ma wartość 0, gdy obiekt v_clob1 jest porównywany z samym sobą, co oznacza, że dane w obiekcie LOB są identyczne.

Kopiowanie danych z jednego obiektu LOB do innego

Poniższa procedura copy_example() kopiuje za pomocą metody COPY() kilka znaków z obiektu v_src_clob do v_dest_clob:

```
CREATE PROCEDURE copy_example AS
  v_src_clob CLOB;
  v_dest_clob CLOB;
  v_src_offset INTEGER := 1;
  v_dest_offset INTEGER := 7;
  v_amount INTEGER := 5;
BEGIN
  -- pobiera do v_src_clob lokalizator LOB obiektu CLOB znajdującego się
  -- w wierszu 2 tabeli clob_content table
  get_clob_locator(v_src_clob, 2);

  -- pobiera do v_dest_clob lokalizator LOB obiektu CLOB znajdującego się
  -- w wierszu 1 tabeli clob_content (w celu aktualizacji,
  -- ponieważ później zostaną dodane dane za pomocą metody COPY())
  SELECT clob_column
  INTO v_dest_clob
  FROM clob_content
  WHERE id = 1
  FOR UPDATE;

  -- czyta i wyświetla zawartość CLOB nr 1
  read_clob_example(1);

  -- za pomocą COPY() kopiuje v_amount znaków z v_src_clob do v_dest_clob,
  -- rozpoczynając w przesunięciach określonych przez v_dest_offset i
  -- v_src_offset
  DBMS_LOB.COPY(
    v_dest_clob, v_src_clob,
    v_amount, v_dest_offset, v_src_offset
  );

  -- czyta i wyświetla zawartość CLOB nr 1,
  -- a następnie wycofuje zmianę
  read_clob_example(1);
  ROLLBACK;
END copy_example;
/
```

Poniżej przedstawiono wynik wywołania procedury copy_example():

```
CALL copy_example();
v_char_buffer = Wije się w ciasnym kółku
v_amount = 24
v_char_buffer = Wije s od dciasnym kółku
v_amount = 24
```

Użycie tymczasowych obiektów CLOB

Poniższa procedura temporary_lob_example() obrazuje użycie tymczasowego obiektu CLOB:

```

CREATE PROCEDURE temporary_lob_example AS
  v_clob CLOB;
  v_amount INTEGER;
  v_offset INTEGER := 1;
  v_char_buffer VARCHAR2(18) := 'Julia jest Słońcem';
BEGIN
  -- za pomocą CREATETEMPORARY() tworzy tymczasowy obiekt CLOB o nazwie v_clob
  DBMS_LOB.CREATETEMPORARY(v_clob, TRUE);

  -- za pomocą WRITE() zapisuje zawartość v_char_buffer do v_clob
  v_amount := LENGTH(v_char_buffer);
  DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

  -- za pomocą ISTEMPORARY() sprawdza, czy v_clob jest obiektem tymczasowym
  IF (DBMS_LOB.ISTEMPORARY(v_clob) = 1) THEN
    DBMS_OUTPUT.PUT_LINE('v_clob jest obiektem tymczasowym');
  END IF;

  -- za pomocą READ() czyta zawartość v_clob do v_char_buffer
  DBMS_LOB.READ(
    v_clob, v_amount, v_offset, v_char_buffer
  );
  DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);

  -- za pomocą FREETEMPORARY() zwalnia v_clob
  DBMS_LOB.FREETEMPORARY(v_clob);
END temporary_lob_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `temporary_lob_example()`:

```

CALL temporary_lob_example();
v_clob jest obiektem tymczasowym
v_char_buffer = Julia jest Słońcem

```

Usuwanie danych z obiektu CLOB

Poniższa procedura `erase_example()` usuwa za pomocą metody ERASE fragment obiektu CLOB:

```

CREATE PROCEDURE erase_example IS
  v_clob CLOB;
  v_offset INTEGER := 2;
  v_amount INTEGER := 5;
BEGIN
  -- pobiera do v_dest_clob lokalizator LOB obiektu CLOB
  -- znajdującego się w wierszu nr 1 tabeli clob_content table
  -- (w celu modyfikacji, ponieważ CLOB zostanie później usunięty)
  -- za pomocą metody ERASE()
  SELECT clob_column
  INTO v_clob
  FROM clob_content
  WHERE id = 1
  FOR UPDATE;

  -- czyta i wyświetla zawartość CLOB nr 1
  read_clob_example(1);

  -- za pomocą ERASE() usuwa v_amount znaków
  -- z v_clob, rozpoczynając w v_offset
  DBMS_LOB.ERASE(v_clob, v_amount, v_offset);

  -- czyta i wyświetla zawartość CLOB nr 1,
  -- a następnie wycofuje zmianę
  read_clob_example(1);

```

```

ROLLBACK;
END erase_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `erase_example()`:

```

CALL erase_example();
v_char_buffer = Wije się w ciasnym kółku
v_amount = 24
v_char_buffer = W      ię w ciasnym kółku
v_amount = 24

```

Wyszukiwanie danych w obiekcie CLOB

W poniższej procedurze `instr_example()` użyto metody `INSTR()` do wyszukania znaku składowanego w obiekcie CLOB:

```

CREATE PROCEDURE instr_example AS
  v_clob CLOB;
  v_char_buffer VARCHAR2(50) := ' Ono jest wschodem, a Julia jest słońcem';
  v_pattern NVARCHAR2(7);
  v_offset INTEGER := 1;
  v_amount INTEGER;
  v_occurrence INTEGER;
  v_return INTEGER;
BEGIN
  -- za pomocą CREATETEMPORARY() tworzy tymczasowy obiekt CLOB o nazwie v_clob
  DBMS_LOB.CREATETEMPORARY(v_clob, TRUE);

  -- za pomocą WRITE() zapisuje zawartość v_char_buffer do v_clob
  v_amount := LENGTH(v_char_buffer);
  DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

  -- za pomocą READ() odczytuje zawartość v_clob do v_char_buffer
  DBMS_LOB.READ(v_clob, v_amount, v_offset, v_char_buffer);
  DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);

  -- za pomocą INSTR() wyszukuje w v_clob drugie wystąpienie słowa "jest",
  -- INSTR() zwraca 29
  DBMS_OUTPUT.PUT_LINE('Wyszukiwanie drugiego "jest"');
  v_pattern := 'jest';
  v_occurrence := 2;
  v_return := DBMS_LOB.INSTR(v_clob, v_pattern, v_offset, v_occurrence);
  DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

  -- za pomocą INSTR() wyszukuje w v_clob pierwsze wystąpienie słowa "Książyc"
  -- INSTR() zwraca 0, ponieważ "Książyc" nie występuje w v_clob
  DBMS_OUTPUT.PUT_LINE('Wyszukiwanie słowa "Książyc"');
  v_pattern := 'Książyc';
  v_occurrence := 1;
  v_return := DBMS_LOB.INSTR(v_clob, v_pattern, v_offset, v_occurrence);
  DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

  -- zwalnia v_clob za pomocą FREETEMPORARY()
  DBMS_LOB.FREETEMPORARY(v_clob);
END instr_example;
/

```

Poniżej przedstawiono wynik wywołania procedury `instr_example()`:

```

CALL instr_example();
v_char_buffer = Ono jest wschodem, a Julia jest słońcem
Wyszukiwanie drugiego 'jest'
v_return = 29
Wyszukiwanie słowa 'Książyc'
v_return = 0

```

Kopiowanie danych z pliku do obiektów CLOB i BLOB

Poniższa procedura `copy_file_data_to_clob()` prezentuje sposób odczytywania tekstu z pliku i składowania go w obiekcie CLOB:

```

CREATE OR REPLACE PROCEDURE copy_file_data_to_clob(
    p_clob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_file UTL_FILE.FILE_TYPE;
    v_chars_read INTEGER;
    v_dest_clob CLOB;
    v_amount INTEGER := 32767;
    v_char_buffer NVARCHAR2(32767);
BEGIN
    -- wstawia pusty obiekt CLOB
    INSERT INTO clob_content(
        id, clob_column
    ) VALUES (
        p_clob_id, EMPTY_CLOB()
    );
    -- pobiera lokalizator LOB obiektu CLOB
    SELECT clob_column
    INTO v_dest_clob
    FROM clob_content
    WHERE id = p_clob_id
    FOR UPDATE;
    -- otwiera plik do odczytu tekstu (do v_amount znaków w wierszu)
    v_file := UTL_FILE.FOPEN(p_directory, p_file_name, 'r', v_amount);
    -- kopiuje dane z pliku do v_dest_clob po jednej linii
    LOOP
        BEGIN
            -- odczytuje wiersz z pliku do v_char_buffer;
            -- GET_LINE() nie kopiuje znaku nowego wiersza
            -- do v_char_buffer
            UTL_FILE.GET_LINE(v_file, v_char_buffer);
            v_chars_read := LENGTH(v_char_buffer);
            -- dodaje wiersz do v_dest_clob
            DBMS_LOB.WRITEAPPEND(v_dest_clob, v_chars_read, v_char_buffer);
            -- dodaje znak nowego wiersza do v_dest_clob, ponieważ v_char_buffer go nie zawiera;
            -- kod ASCII znaku nowego wiersza to 10, więc CHR(10) zwraca ten znak
            DBMS_LOB.WRITEAPPEND(v_dest_clob, 1, CHR(10));
        EXCEPTION
            -- jeżeli w pliku nie ma więcej danych, zakończ działanie
            WHEN NO_DATA_FOUND THEN
                EXIT;
        END;
    END LOOP;
    -- zamyka plik
    UTL_FILE.FCLOSE(v_file);
    DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończyło się powodzeniem.');
END copy_file_data_to_clob;
/

```

Należy zwrócić uwagę na kilka aspektów tej procedury:

- UTL_FILE jest pakietem dostarczonym z bazą danych, który zawiera metody i typy umożliwiające odczytywanie i zapisywanie plików. Na przykład UTL_FILE.FILE_TYPE jest typem obiektowym reprezentującym plik.
- Zmiennej v_amount jest przypisywana wartość 32 767, czyli maksymalna liczba znaków, która może być odczytana z pliku podczas jednej operacji odczytu.
- W zmiennej v_char_buffer są składowane wyniki odczytu z pliku, zanim zostaną dołączone do v_dest_clob. Maksymalna długość v_char_buffer została ustawiona na 32 767, co wystarczy do pomieszczenia maksymalnej liczby znaków odczytywanych z pliku podczas każdej operacji odczytu.
- Metoda UTL_FILE.FOPEN(*katalog, nazwa_pliku, tryb, liczba*) otwiera plik. Parametr *tryb* może mieć jedną z poniższych wartości:
 - r dla czytania tekstu,
 - w dla zapisu tekstu,
 - a dla dołączania tekstu,
 - rb dla odczytu bajtów,
 - wb dla zapisu bajtów,
 - ab dla dołączania bajtów.
- Metoda UTL_FILE.GET_LINE(*v_file, v_char_buffer*) pobiera wiersz tekstu z *v_file* do *v_char_buffer*. Metoda GET_LINE() nie wstawia znaku nowego wiersza do *v_char_buffer*, a ponieważ jest nam on potrzebny, wstawiamy go, używając wywołania DBMS_LOB.WRITEAPPEND(*v_dest_clob, 1, CHR(10)*).

W poniższym przykładzie wywołano metodę *copy_file_data_to_clob()* w celu skopiowania zawartości pliku *textContent.txt* do nowego obiektu CLOB o id wynoszącym 3:

```
CALL copy_file_data_to_clob(3, 'SAMPLE_FILES_DIR', 'textContent.txt');
```

Kopiowanie zakończyło się powodzeniem.

Poniższa procedura *copy_file_data_to_blob()* obrazuje sposób odczytu danych binarnych z pliku i składowania ich w obiekcie BLOB. Należy zauważyć, że do składowania danych binarnych odczytanych z pliku używana jest tablica RAW:

```
CREATE PROCEDURE copy_file_data_to_blob(
  p_blob_id INTEGER,
  p_directory VARCHAR2,
  p_file_name VARCHAR2
) AS
  v_file UTL_FILE.FILE_TYPE;
  v_bytes_read INTEGER;
  v_dest_blob BLOB;
  v_amount INTEGER := 32767;
  v_binary_buffer RAW(32767);
BEGIN
  -- wstawia pusty obiekt BLOB
  INSERT INTO blob_content(
    id, blob_column
  ) VALUES (
    p_blob_id, EMPTY_BLOB()
  );

  -- pobiera lokalizator LOB obiektu BLOB
  SELECT blob_column
  INTO v_dest_blob
  FROM blob_content
  WHERE id = p_blob_id
  FOR UPDATE;

  -- otwiera plik w trybie odczytu bajtów (do v_amount bajtów w jednej operacji)
  v_file := UTL_FILE.FOPEN(p_directory, p_file_name, 'rb', v_amount);
```

```
-- kopiuje dane z pliku do v_dest_blob
LOOP
BEGIN
    -- czyta dane binarne z pliku do v_binary_buffer
    UTL_FILE.GET_RAW(v_file, v_binary_buffer, v_amount);
    v_bytes_read := LENGTH(v_binary_buffer);

    -- dodaje v_binary_buffer do v_dest_blob
    DBMS_LOB.WRITEAPPEND(v_dest_blob, v_bytes_read/2,
    v_binary_buffer);
EXCEPTION
    -- zakończ, jeżeli nie ma więcej danych w pliku
    WHEN NO_DATA_FOUND THEN
        EXIT;
    END;
END LOOP;

-- zamyka plik
UTL_FILE.FCLOSE(v_file);

DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończone powodzeniem.');
END copy_file_data_to_blob;
/
```

Poniżej przedstawiono wywołanie procedury `copy_file_data_to_blob()` w celu skopiowania zawartości pliku `binaryContent.doc` do nowego obiektu BLOB z id wynoszącym 3:

```
CALL copy_file_data_to_blob(3, 'SAMPLE_FILES_DIR', 'binaryContent.doc');
Kopiowanie zakończone powodzeniem.
```

Za pomocą procedury `copy_file_data_to_blob()` można skopiować do obiektu BLOB wszelkie dane binarne znajdujące się w pliku. Danymi binarnymi są na przykład muzyka, filmy, obrazy i pliki wykonywalne. Wypróbuj ją również, używając swoich własnych plików.



Dane można również ładować masowo do obiektu LOB za pomocą narzędzi Oracle SQL*Loader i Data Pump. Informacje na ten temat można znaleźć w podręczniku *Oracle Database Large Objects Developer's Guide* opublikowanym przez Oracle Corporation.

Kopiowanie danych z obiektów CLOB i BLOB do pliku

Poniższa procedura `copy_clob_data_to_file()` pokazuje, jak można odczytać tekst z obiektu CLOB i zapisać go w pliku:

```
CREATE PROCEDURE copy_clob_data_to_file(
    p_clob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_src_clob CLOB;
    v_file UTL_FILE.FILE_TYPE;
    v_offset INTEGER := 1;
    v_amount INTEGER := 32767;
    v_char_buffer NVARCHAR2(32767);
BEGIN
    -- pobiera lokalizator LOB obiektu CLOB
    SELECT clob_column
    INTO v_src_clob
    FROM clob_content
    WHERE id = p_clob_id;

    -- otwiera plik w trybie zapisu tekstu (do v_amount znaków w operacji)
    v_file := UTL_FILE.FOPEN(p_directory, p_file_name, 'w', v_amount);

    -- kopiuje dane z v_src_clob do pliku
    LOOP
```

```

BEGIN
  -- czyta znaki z v_src_clob do v_char_buffer
  DBMS_LOB.READ(v_src_clob, v_amount, v_offset, v_char_buffer);

  -- kopiuje znaki z v_char_buffer do pliku
  UTL_FILE.PUT(v_file, v_char_buffer);

  -- dodaje v_amount do v_offset
  v_offset := v_offset + v_amount;
EXCEPTION
  -- zakończ, jeżeli nie ma więcej danych w pliku
  WHEN NO_DATA_FOUND THEN
    EXIT;
END;
END LOOP;

-- zapisz pozostałe dane do pliku
UTL_FILE.FFLUSH(v_file);

-- zamknij plik
UTL_FILE.FCLOSE(v_file);

DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończyło się powodzeniem.');
END copy_clob_data_to_file;
/

```

Poniżej przedstawiono wywołanie procedury `copy_clob_data_to_file()` w celu skopiowania zawartości obiektu CLOB nr 3 do pliku o nazwie `textContent2.txt`:

```
CALL copy_clob_data_to_file(3, 'SAMPLE_FILES_DIR', 'textContent2.txt');
Kopiowanie zakończyło się powodzeniem.
```

Działanie procedury można zweryfikować — w katalogu `C:\sample_files` powinien się znajdować nowy plik `textContent2.txt`. Zawiera on taki sam tekst jak `textContent.txt`.

Poniższa procedura `copy_blob_data_to_file()` stanowi przykład odczytu danych z obiektu BLOB i zapisania ich do pliku:

```

CREATE PROCEDURE copy_blob_data_to_file(
  p_blob_id INTEGER,
  p_directory VARCHAR2,
  p_file_name VARCHAR2
) AS
  v_src_blob BLOB;
  v_file UTL_FILE.FILE_TYPE;
  v_offset INTEGER := 1;
  v_amount INTEGER := 32767;
  v_binary_buffer RAW(32767);
BEGIN
  -- pobiera lokalizator LOB obiektu BLOB
  SELECT blob_column
  INTO v_src_blob
  FROM blob_content
  WHERE id = p_blob_id;

  -- otwiera plik do zapisu bajtów (do v_amount bajtów w operacji)
  v_file := UTL_FILE.FOPEN(p_directory, p_file_name, 'wb', v_amount);

  -- kopiuje dane z v_src_blob do pliku
LOOP
  BEGIN
    -- czyta dane binarne z v_src_blob do v_binary_buffer
    DBMS_LOB.READ(v_src_blob, v_amount, v_offset, v_binary_buffer);

    -- kopiuje dane binarne z v_binary_buffer do pliku
    UTL_FILE.PUT_RAW(v_file, v_binary_buffer);
  END;
END;

```

```

-- dodaje v_amount do v_offset
v_offset := v_offset + v_amount;
EXCEPTION
  -- jeżeli nie ma więcej danych, wychodzi z pętli
  WHEN NO_DATA_FOUND THEN
    EXIT;
  END;
END LOOP;

-- zapisuje pozostałe dane do pliku
UTL_FILE.FFLUSH(v_file);

-- zamyka plik
UTL_FILE.FCLOSE(v_file);

DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończone powodzeniem.');
END copy_blob_data_to_file;
/

```

Poniżej przedstawiono wywołanie procedury `copy_blob_data_to_file()` w celu skopiowania zawartości obiektu BLOB nr 3 do pliku o nazwie `binaryContent2.doc`:

```
CALL copy_blob_data_to_file(3, 'SAMPLE_FILES_DIR', 'binaryContent2.doc');
Kopiowanie zakończone powodzeniem.
```

Działanie procedury można zweryfikować — w katalogu `C:\sample_files` powinien się znajdować nowy plik `binaryContent2.doc`. Zawiera on taki sam tekst jak `binaryContent.doc`.

Za pomocą procedury `copy_blob_data_to_file()` można skopiać do pliku dowolne dane składowane w obiekcie BLOB. Dane binarne mogą zawierać muzykę, film, obrazy, kod wykonywalny itd.

Kopiowanie danych z obiektu BFILE do CLOB i BLOB

Poniższa procedura `copy_bfile_data_to_clob()` odczytuje tekst z obiektu BFILE i zapisuje go w obiekcie CLOB:

```

CREATE PROCEDURE copy_bfile_data_to_clob(
  p_bfile_id INTEGER,
  p_clob_id INTEGER
) AS
  v_src_bfile BFILE;
  v_directory VARCHAR2(200);
  v_filename VARCHAR2(200);
  v_length INTEGER;
  v_dest_clob CLOB;
  v_amount INTEGER := DBMS_LOB.LOBMAXSIZE;
  v_dest_offset INTEGER := 1;
  v_src_offset INTEGER := 1;
  v_src_csid INTEGER := DBMS_LOB.DEFAULT_CSID;
  v_lang_context INTEGER := DBMS_LOB.DEFAULT_LANG_CTX;
  v_warning INTEGER;
BEGIN
  -- pobiera lokalizator obiektu BFILE
  SELECT bfile_column
  INTO v_src_bfile
  FROM bfile_content
  WHERE id = p_bfile_id;

  -- za pomocą FILE EXISTS() sprawdza, czy plik istnieje
  -- (FILE EXISTS() zwraca 1, jeśli plik istnieje)
  IF (DBMS_LOB.FILE EXISTS(v_src_bfile)) = 1 THEN
    -- otwiera plik za pomocą OPEN()
    DBMS_LOB.OPEN(v_src_bfile);

    -- pobiera nazwę pliku i katalogu za pomocą FILE GETNAME()
    DBMS_LOB.FILE GETNAME(v_src_bfile, v_directory, v_filename);

```

```

DBMS_OUTPUT.PUT_LINE('Katalog = ' || v_directory);
DBMS_OUTPUT.PUT_LINE('Nazwa pliku = ' || v_filename);

-- wstawia pusty obiekt CLOB
INSERT INTO clob_content(
    id, clob_column
) VALUES (
    p_clob_id, EMPTY_CLOB()
);

-- pobiera lokalizator LOB obiektu CLOB (w celu modyfikacji obiektu)
SELECT clob_column
INTO v_dest_clob
FROM clob_content
WHERE id = p_clob_id
FOR UPDATE;

-- za pomocą LOADCLOBFROMFILE() pobiera z v_src_bfile maksymalnie v_amount znaków
-- i zapisuje je w v_dest_clob (rozpoczyna w przesunięciach l w v_src_bfile i v_dest_clob
DBMS_LOB.LOADCLOBFROMFILE(
    v_dest_clob, v_src_bfile,
    v_amount, v_dest_offset, v_src_offset,
    v_src_csid, v_lang_context, v_warning
);

-- sprawdza w v_warning, czy wystąpił niekonwertowalny znak
IF (v_warning = DBMS_LOB.WARN_INCONVERTIBLE_CHAR) THEN
    DBMS_OUTPUT.PUT_LINE('Uwaga! Konwersja znaku nie powiodła się.');
END IF;

-- zamyka v_src_bfile za pomocą CLOSE()
DBMS_LOB CLOSE(v_src_bfile);
DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończone powodzeniem.');

ELSE
    DBMS_OUTPUT.PUT_LINE('Plik nie istnieje.');
END IF;
END copy_bfile_data_to_clob;
/

```

Poniżej przedstawiono wywołanie procedury `copy_bfile_data_to_clob()` w celu skopiowania zawartości BFILE nr 1 do nowego obiektu CLOB o identyfikatorze 4:

```

CALL copy_bfile_data_to_clob(1, 4);
Katalog = SAMPLE_FILES_DIR
Nazwa pliku = textContent.txt
Kopiowanie zakończone powodzeniem.

```

Kolejny przykład przedstawia wywołanie procedury `copy_clob_data_to_file` w celu skopiowania zawartości obiektu CLOB nr 4 do nowego pliku o nazwie `textContent3.txt`:

```

CALL copy_clob_data_to_file(4, 'SAMPLE_FILES_DIR', 'textContent3.txt');
Kopiowanie zakończyło się powodzeniem.

```

W katalogu `C:\sample_files` został utworzony nowy plik o nazwie `textContent3.txt`, którego zawartość jest taka sama jak pliku `textContent.txt`.

Poniższa procedura `copy_bfile_data_to_blob()` przedstawia sposób odczytu danych binarnych z BFILE i zapisywania ich w obiekcie BLOB:

```

CREATE PROCEDURE copy_bfile_data_to_blob(
    p_bfile_id INTEGER,
    p_blob_id INTEGER
) AS
    v_src_bfile BFILE;
    v_directory VARCHAR2(200);
    v_filename VARCHAR2(200);
    v_length INTEGER;

```

```

v_dest_blob BLOB;
v_amount INTEGER := DBMS_LOB.LOBMAXSIZE;
v_dest_offset INTEGER := 1;
v_src_offset INTEGER := 1;
BEGIN
  -- pobiera lokalizator obiektu BFILE
  SELECT bfile_column
  INTO v_src_bfile
  FROM bfile_content
  WHERE id = p_bfile_id;

  -- za pomocą FILEEXISTS() sprawdza, czy plik istnieje
  -- (FILEEXISTS() zwraca 1, jeżeli plik istnieje)
  IF (DBMS_LOB.FILEEXISTS(v_src_bfile) = 1) THEN
    -- otwiera plik za pomocą OPEN()
    DBMS_LOB.OPEN(v_src_bfile);

    -- pobiera nazwę pliku i katalogu za pomocą FILEGETNAME()
    DBMS_LOB.FILEGETNAME(v_src_bfile, v_directory, v_filename);
    DBMS_OUTPUT.PUT_LINE('Katalog = ' || v_directory);
    DBMS_OUTPUT.PUT_LINE('Nazwa pliku = ' || v_filename);

    -- wstawia pusty obiekt BLOB
    INSERT INTO blob_content(
      id, blob_column
    ) VALUES (
      p_blob_id, EMPTY_BLOB()
    );

    -- pobiera lokalizator LOB nowego obiektu BLOB (w celu modyfikacji obiektu)
    SELECT blob_column
    INTO v_dest_blob
    FROM blob_content
    WHERE id = p_blob_id
    FOR UPDATE;

    -- za pomocą LOADBLOBFROMFILE() pobiera z v_src_bfile maksymalnie v_amount bajtów
    -- i zapisuje je w v_dest_blob (rozpoczyna w przesunięciach 1 w v_src_bfile i v_dest_blob)
    DBMS_LOBLOADBLOBFROMFILE(
      v_dest_blob, v_src_bfile,
      v_amount, v_dest_offset, v_src_offset
    );

    -- zamyka v_src_bfile za pomocą CLOSE()
    DBMS_LOB.CLOSE(v_src_bfile);
    DBMS_OUTPUT.PUT_LINE('Kopiowanie zakończone powodzeniem.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Plik nie istnieje.');
  END IF;
END copy_bfile_data_to_blob;
/

```

Poniżej przedstawiono wywołanie procedury `copy_bfile_data_to_blob()` w celu skopiowania zawartości BFILE nr 2 do nowego obiektu BLOB o identyfikatorze 4:

```

CALL copy_bfile_data_to_blob(2, 4);
Katalog = SAMPLE_FILES_DIR
Nazwa pliku = textContent.txt
Kopiowanie zakończone powodzeniem.

```

Kolejny przykład przedstawia wywołanie procedury `copy_blob_data_to_file` w celu skopiowania zawartości obiektu BLOB nr 4 do nowego pliku o nazwie `binaryContent3.doc`:

```

CALL copy_blob_data_to_file(4, 'SAMPLE_FILES_DIR', 'binaryContent3.doc');
Kopiowanie zakończyło się powodzeniem.

```

W katalogu C:\sample_files został utworzony nowy plik o nazwie *binaryContent3.doc*, którego zawartość jest taka sama jak pliku *binaryContent.doc*.

Na tym zakończymy opis dużych obiektów. W kolejnym podrozdziale poznasz typy LONG i LONG RAW.

Typy LONG i LONG RAW

Na początku tego rozdziału wspomniano, że preferowanym typem dla składowania dużych bloków danych są typy LOB. Wciąż jednak można napotkać bazy danych, w których używane są następujące typy:

- **LONG** — używany do składowania maksymalnie 2 gigabajtów danych znakowych,
- **LONG RAW** — używany do składowania maksymalnie 2 gigabajtów danych binarnych,
- **RAW** — używany do składowania maksymalnie 4 kilobajtów danych binarnych.

Z tego podrozdziału dowiesz się, jak używać typów LONG i LONG RAW. Typ RAW jest używany w taki sam sposób jak typ LONG RAW, jego opis został więc pominięty.

Przykładowe tabele

W tym podrozdziale będziemy używali dwóch tabel:

- **long_content** zawierającej kolumnę typu LONG o nazwie `long_column`,
- **long_raw_content** zawierającej kolumnę typu LONG RAW o nazwie `long_raw_column`.

Te dwie tabele są tworzone przez skrypt *lob_schema.sql* za pomocą poniższych instrukcji:

```
CREATE TABLE long_content (
    id INTEGER PRIMARY KEY,
    long_column LONG NOT NULL
);

CREATE TABLE long_raw_content (
    id INTEGER PRIMARY KEY,
    long_raw_column LONG RAW NOT NULL
);
```

Wstawianie danych do kolumn typu LONG i LONG RAW

Poniższe instrukcje wstawiają wiersze do tabeli `long_content`:

```
INSERT INTO long_content (
    id, long_column
) VALUES (
    1, ' Wije się w ciasnym kółku'
);

INSERT INTO long_content (
    id, long_column
) VALUES (
    2, ' od dnia do dnia'
);
```

Poniższe instrukcje `INSERT` wstawiają wiersze do tabeli `long_raw_content` (pierwsza z instrukcji zawiera liczbę w systemie dwójkowym, a druga — w szesnastkowym):

```
INSERT INTO long_raw_content (
    id, long_raw_column
) VALUES (
    1, '100111010101011111'
);

INSERT INTO long_raw_content (
    id, long_raw_column
)
```

```
) VALUES (
 2, 'A0FFB71CF90DE'
);
```

W następnym podrozdziale dowiesz się, jak przekształcić kolumny typu LONG i LONG RAW w duże obiekty.

Przekształcanie kolumn LONG i LONG RAW w duże obiekty

Za pomocą funkcji `TO_LOB()` można przekształcić typ LONG w CLOB. Na przykład poniższa instrukcja, korzystając z funkcji `TO_LOB`, konwertuje kolumnę `long_column` w CLOB i zapisuje wyniki do tabeli `clob_content`:

```
INSERT INTO clob_content
SELECT 10 + id, TO_LOB(long_column)
FROM long_content;
```

2 rows created.

Za pomocą funkcji `TO_LOB()` można również przekształcić typ LONG RAW w BLOB. Na przykład poniższa instrukcja konwertuje za pomocą tej funkcji kolumnę `long_raw_column` na BLOB i zapisuje wyniki w tabeli `blob_content`:

```
INSERT INTO blob_content
SELECT 10 + id, TO_LOB(long_raw_column)
FROM long_raw_content;
```

2 rows created.

Za pomocą instrukcji `ALTER TABLE` można bezpośrednio przekonwertować kolumny LONG i LONG RAW. Na przykład poniższa instrukcja konwertuje `long_column` na CLOB:

```
ALTER TABLE long_content MODIFY (long_column CLOB);
```

Kolejny przykład konwertuje kolumnę `long_raw_column` na BLOB:

```
ALTER TABLE long_raw_content MODIFY (long_raw_column BLOB);
```

Po skonwertowaniu kolumn LONG i LONG RAW na LOB można pracować z nimi za pomocą przedstawionego wcześniej, szerokiego zestawu metod PL/SQL.

Nowe właściwości dużych obiektów w Oracle Database 10g

W tym podrozdziale poznasz wprowadzone w Oracle Database 10g właściwości dużych obiektów:

- niejawną konwersję między obiektami CLOB i NCLOB,
- użycie atrybutu `:new`, jeżeli LOB jest używany w wyzwalaczku.

W katalogu `SQL` znajduje się skrypt `SQL*Plus` o nazwie `lob_schema2.sql`. Może on zostać uruchomiony w Oracle Database 10g lub nowszej. Ten skrypt tworzy konto użytkownika o nazwie `lob_user2` z hasłem `lob_password2`, a także wszystkie tabele i kod wykorzystywany w tym podrozdziale.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić `SQL*Plus`.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika `system`.
3. Uruchomić skrypt `lob_schema2.sql` w `SQL*Plus` za pomocą polecenia `@`.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu `C:\SQL`, to należy wpisać polecenie:

```
@ C:\SQL\lob_schema2.sql
```

Po zakończeniu pracy skryptu jest zalogowany użytkownik `lob_user2`.

Niejawnna konwersja między obiektami CLOB i NCLOB

We współczesnym globalnym środowisku biznesowym może być wymagana konwersja między zestawem Unicode i zestawami znaków narodowych. Unicode jest uniwersalnym zestawem znaków, umożliwiającym składowanie tekstu, który może zostać skonwertowany na dowolny język. Jest to możliwe, ponieważ w Unicode każdy znak, niezależnie od języka, posiada unikatowy kod. Zestaw znaków językowych składa się z tekstu w konkretnym języku.

W wersjach wcześniejszych niż Oracle Database 10g konwersja między tekstem Unicode i zestawem znaków narodowych musiała być przeprowadzana jawnie, za pomocą funkcji `TO_CLOB()` i `TO_NCLOB()`. Funkcja `TO_CLOB()` umożliwia konwersję tekstu składowanego jako `VARCHAR2`, `NVARCHAR2` i `NCLOB` na `CLOB`, a funkcja `TO_NCLOB()` umożliwia konwersję tekstu składowanego jako `VARCHAR2`, `NVARCHAR2` i `CLOB` na `NCLOB`.

W Oracle Database 10g i nowszych konwersja tekstu Unicode i zestawów narodowych odbywa się niejawnie, co pozwala uniknąć stosowania funkcji `TO_CLOB()` i `TO_NCLOB()`. Tę niejawną konwersję można wykorzystać w zmiennych `IN` i `OUT` w zapytaniach i instrukcjach DML, a także w parametrach i przy pisywaniu wartości zmiennych w kodzie PL/SQL.

Przejedźmy do przykładu. Poniższa instrukcja tworzy tabelę o nazwie `nclob_content`, zawierającą kolumnę typu `NCLOB` o nazwie `nclob_column`:

```
CREATE TABLE nclob_content (
    id INTEGER PRIMARY KEY,
    nclob_column NCLOB
);
```

Poniższa procedura `nclob_example()` obrazuje niejawną konwersję obiektu `CLOB` na `NCLOB` i odwrotnie:

```
CREATE OR REPLACE PROCEDURE nclob_example
AS
    clob_var CLOB := 'To jest tekst CLOB';
    nclob_var NCLOB;
BEGIN
    -- wstawia clob_var do nclob_column, co powoduje niejawną konwersję
    -- obiektu v_clob typu CLOB na NCLOB i zachowanie zawartości
    -- v_clob w tabeli nclob_content
    INSERT INTO nclob_content (
        id, nclob_column
    ) VALUES (
        1, clob_var
    );
    -- wybiera nclob_column do clob_var, co powoduje niejawną konwersję
    -- obiektu NCLOB składowanego w kolumnie nclob_column
    -- na CLOB i pobranie zawartości nclob_column do v_clob
    SELECT nclob_column
    INTO clob_var
    FROM nclob_content
    WHERE id = 1;
    -- wyświetla zawartość CLOB
    DBMS_OUTPUT.PUT_LINE('clob_var = ' || clob_var);
END nclob_example;
/
```

Poniższy przykład włącza wypisywanie z serwera i wywołuje procedurę `nclob_example()`:

```
SET SERVEROUTPUT ON
CALL nclob_example();
clob_var = To jest tekst CLOB
```

Użycie atrybutu :new, gdy obiekt LOB jest używany w wyzwalaczu

W Oracle Database 10g można użyć atrybutu `:new`, odwołując się do obiektu LOB w wyzwalaczu `BEFORE UPDATE` lub `BEFORE INSERT` poziomu wiersza. Poniższy przykład tworzy wyzwalacz o nazwie `before_clob_`

`content_update`. Jest uruchamiany, gdy tabela `clob_content` jest modyfikowana, i wyświetla długość nowych danych w kolumnie `clob_column`. Należy zauważać użycie atrybutu `:new` do uzyskania dostępu do nowych danych w `clob_column`:

```
CREATE OR REPLACE TRIGGER before_clob_content_update
BEFORE UPDATE
ON clob_content
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('zawartość clob_content została zmieniona');
  DBMS_OUTPUT.PUT_LINE(
    'Długość = ' || DBMS_LOB.GETLENGTH(:new.clob_column)
  );
END before_clob_content_update;
/
```

Poniższy przykład modyfikuje tabelę `clob_content`, co powoduje uruchomienie wyzwalacza:

```
UPDATE clob_content
SET clob_column = 'Wije się w ciasnym kółku'
WHERE id = 1;
zawartość clob_content została zmieniona
Długość = 24
```

Nowe właściwości dużych obiektów w Oracle Database 11g

W tym podrozdziale poznasz nowe właściwości dużych obiektów, wprowadzone w Oracle Database 11g:

- szyfrowanie danych `BLOB`, `CLOB` i `NCLOB` zapobiegające nieuprawnionemu dostępowi i modyfikowaniu tych danych,
- kompresję zmniejszającą rozmiar danych `BLOB`, `CLOB` i `NCLOB`,
- deduplikację danych `BLOB`, `CLOB` i `NCLOB` automatycznie wykrywającą i usuwającą powtarzające się dane.

Szyfrowanie danych LOB

Z pomocą szyfrowania można utajnić dane, dzięki czemu nieuprawnieni użytkownicy nie będą mogli ich przeglądać i modyfikować. Należy szyfrować dane wrażliwe, takie jak numery kart kredytowych, numery PESEL itd.

Zanim będzie możliwe szyfrowanie danych, należy skonfigurować „portfel”, w którym będą składowane informacje o zabezpieczeniach. Dane umieszczane w portfelu zawierają klucz prywatny do szyfrowania i deszyfrowania danych. Z tego podrozdziału dowiesz się, jak utworzyć portfel, zaszyfrować dane LOB oraz dane zwykłej kolumny.

Tworzenie portfela

W celu utworzenia portfela należy:

1. Utworzyć katalog `wallet` w katalogu `$ORACLE_BASE\admin\$ORACLE_SID`, gdzie `ORACLE_BASE` jest podstawowym katalogiem, w którym jest zainstalowane oprogramowanie Oracle, a `ORACLE_SID` jest identyfikatorem systemowym bazy danych, dla której będzie tworzony portfel. Na przykład w systemie Windows, katalog `wallet` można utworzyć w katalogu `C:\oracle12c\admin\orcl`. W systemie Linux można taki katalog utworzyć w `/u01/app/oracle12c/admin/orcl`.
2. Wyedytować plik `sqlnet.ora` znajdujący się w podkatalogu `NETWORK\ADMIN` (np. `C:\oracle12c\product\12.1.0\dbhome_1\NETWORK\ADMIN`) i dodać na końcu pliku linię wskazującą na katalog z portfelem. Na przykład:

```
ENCRYPTION_WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=C:\oracle12c\admin\orcl\wallet)))
```

3. Zrestartować bazę danych Oracle.
4. Uruchomić SQL*Plus, połączyć się z bazą danych jako użytkownik o podwyższonych uprawnieniach (na przykład `system`) i uruchomić instrukcję `ALTER SYSTEM` w celu ustawienia hasła dla klucza szyfrowania:

```
ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "testpassword123";
```

Po zakończeniu wykonywania tej instrukcji w katalogu `wallet` zostanie utworzony plik `ewallet.p12`, a baza danych automatycznie otworzy portfel. Hasło do klucza szyfrującego jest przechowywane w portfelu i automatycznie jest stosowane do szyfrowania i deszyfrowania danych.

W katalogu `SQL` znajduje się skrypt o nazwie `lob_schema3.sql`. Może on być uruchomiony w Oracle Database 11g lub nowszej wersji. Tworzy on konto użytkownika o nazwie `lob_user3` i hasło `lob_password`, a także tabele wykorzystywane w tym podrozdziale.

Aby utworzyć schemat bazy danych, należy wykonać następujące kroki:

1. Uruchomić SQL*Plus.
2. Zalogować się do bazy danych jako użytkownik z uprawnieniami do tworzenia nowych użytkowników, tabel i pakietów PL/SQL. Ja uruchamiam skrypty w mojej bazie danych, używając konta użytkownika `system`.
3. Uruchomić skrypt `lob_schema3.sql` w SQL*Plus za pomocą polecenia `@`.

Jeśli na przykład pracujesz w systemie Windows i skrypt jest zapisany w katalogu `C:\SQL`, to należy wpisać polecenie:

```
@ C:\SQL\lob_schema3.sql
```

Po zakończeniu pracy skryptu zalogowany jest użytkownik `lob_user3`.

Szyfrowanie danych LOB

W celu uniemożliwienia nieuprawnionemu użytkownikowi dostępu do danych składowanych w `BLOB`, `CLOB` i `NCLOB` można zaszyfrować te obiekty. Zaszyfrowanie obiektu `BFILE` nie jest możliwe, ponieważ plik jest składowany poza bazą danych.

Dane mogą zostać zaszyfrowane za pomocą jednego z poniższych algorytmów:

- **3DES168** — potrójnego DES (*Data Encryption Standard*) z kluczem o długości 168 bitów.
- **AES128** — algorytmu *Advanced Encryption Standard* z kluczem o długości 128 bitów. Algorytmy AES zostały opracowane w celu zastąpienia starszych algorytmów opartych na DES.
- **AES192** — algorytmu *Advanced Encryption Standard* z kluczem o długości 192 bitów.
- **AES256** — algorytmu *Advanced Encryption Standard* z kluczem o długości 256 bitów. Jest to najbezpieczniejszy algorytm szyfrujący obsługiwany przez bazę danych Oracle.

Poniższa instrukcja tworzy tabelę typu `CLOB`, której zawartość będzie szyfrowana za pomocą algorytmu AES128. Należy zauważać użycie słów kluczowych `ENCRYPT` i `SECUREFILE`, które są wymagane przy szyfrowaniu danych:

```
CREATE TABLE clob_content (
    id INTEGER PRIMARY KEY,
    clob_column CLOB ENCRYPT USING 'AES128'
) LOB(clob_column) STORE AS SECUREFILE (
    CACHE
);
```

Zawartość kolumny `clob_column` będzie szyfrowana z użyciem algorytmu AES128. Jeżeli zostanie pominięte słowo kluczowe `USING`, zostanie zastosowany domyślny algorytm AES192.

Słowo kluczowe `CACHE` w instrukcji `CREATE TABLE` oznacza, że dane z obiektu LOB będą umieszczane w pamięci podrzędnej w celu przyspieszenia dostępu. Bufor pamięci podrzędnej może zostać użyty za pomocą następujących poleceń:

- **CACHE READS** — stosowanego, gdy dane z obiektu LOB będą często odczytywane, ale zapisywane raz lub bardzo rzadko,
- **CACHE** — używanego, gdy dane z obiektu LOB będą często odczytywane i zapisywane.
- **NOCACHE** — używanego, gdy dane z obiektu LOB będą rzadko odczytywane i zapisywane raz albo rzadko. Jest to opcja domyślna.

Nadaną przez Oracle Corporation oficjalną nazwą obiektów LOB definiowanych z użyciem **SECUREFILE** jest *SecureFiles LOBs*.



Zabezpieczone obiekty LOB (*SecureFiles LOBs*) mogą być tworzone jedynie w przestrzeniach tabel skonfigurowanych za pomocą ASSM (Automatic Segment Space Management). Jeśli pojawią się problemy z tworzeniem tego typu obiektów, powinieneś porozmawiać na ten temat z administratorem Twojej bazy danych.

Poniższe instrukcje **INSERT** wstawiają dwa wiersze do tabeli **clob_content**:

```
INSERT INTO clob_content (
    id, clob_column
) VALUES (
    1, TO_CLOB(' Wije się w ciasnym kółku')
);

INSERT INTO clob_content (
    id, clob_column
) VALUES (
    2, TO_CLOB(' od dnia do dnia')
);
```

Dane, które są wstawiane przez te instrukcje do kolumny **clob_column**, są automatycznie szyfrowane.

Poniższe zapytanie pobiera wiersze z tabeli **clob_content**:

```
SELECT *
FROM clob_content;
```

ID	
1	Wije się w ciasnym kółku
2	od dnia do dnia

Przy pobieraniu dane są automatycznie deszyfrowane przez bazę danych, a następnie zwracane do SQL*Plus.

Jeżeli tylko portfel jest otwarty, można odczytywać i składować zaszyfrowane dane. Jeżeli portfel jest zamknięty, nie jest to możliwe. Zobaczmy, co się stanie po zamknięciu portfela. Poniższe instrukcje nawiążą połączenie jako użytkownik **system** i zamkują portfel:

```
CONNECT system/oracle
ALTER SYSTEM SET WALLET CLOSE IDENTIFIED BY "testpassword123";
```

Jeżeli teraz spróbujemy połączyć się jako użytkownik **lob_user3** i pobrać dane z kolumny **clob_column** tabeli **clob_content**, zostanie zgłoszony błąd ORA-28365: **portfel nie jest otwarty**:

```
CONNECT lob_user3/lob_password
SELECT clob_column
FROM clob_content;
ERROR:
ORA-28365: portfel nie jest otwarty
```

Wciąż jednak można pobierać i modyfikować dane z kolumn niezaszyfrowanych. Na przykład poniższe zapytanie pobiera kolumnę **id** z tabeli **clob_content**:

```
SELECT id
FROM clob_content;

ID
-----
1
2
```

Poniższe instrukcje nawiązują połączenie jako użytkownik system i ponownie otwierają portfel:

```
CONNECT system/oracle
ALTER SYSTEM SET WALLET OPEN IDENTIFIED BY "testpassword123";
```

Po zakończeniu wykonywania tych instrukcji można pobierać i modyfikować zawartość kolumny `clob_column` tabeli `clob_content`.

Szyfrowanie danych kolumny

Można również zaszyfrować dane zwykłej kolumny. Ta możliwość została wprowadzona w Oracle Database Release 2. Na przykład poniższe instrukcje tworzą tabelę o nazwie `credit_cards` z szyfrowaną kolumną `card_number`:

```
CREATE TABLE credit_cards (
    card_number NUMBER(16, 0) ENCRYPT,
    first_name VARCHAR2(10),
    last_name VARCHAR2(10),
    expiration DATE
);
```

Do szyfrowania danych kolumny można stosować te same algorytmy co w przypadku obiektów LOB: 3DES168, AES128, AES192 (domyślnie) i AES256. Ponieważ w powyższej instrukcji nie określono algorytmu po słowie kluczowym `ENCRYPT`, kolumna `card_number` będzie szyfrowana z użyciem domyślnego algorytmu AES192.

Poniższe instrukcje wstawiają dwa wiersze do tabeli `credit_cards`:

```
INSERT INTO credit_cards (
    card_number, first_name, last_name, expiration
) VALUES (
    1234, 'Jan', 'Brązowy', '08/02/03'
);

INSERT INTO credit_cards (
    card_number, first_name, last_name, expiration
) VALUES (
    5768, 'Stefan', 'Czerny', '09/03/07'
);
```

Jeżeli portfel jest otwarty, można pobierać i modyfikować zawartość kolumny `card_number`. Jeśli portfel zostanie zamknięty, będzie zgłoszany błąd ORA-28365: `portfel nie jest otwarty`. Odpowiednie przykłady były prezentowane w poprzednim rozdziale, nie będą więc tutaj powtarzane.

Dostęp do zaszyfrowanych danych wiąże się z pewnym narzutem. Oracle Corporation szacuje, że narzut w przypadku szyfrowania i deszyfrowania kolumn wynosi około 5 procent, co oznacza, że wykonanie instrukcji `SELECT` lub `INSERT` zajmuje około 5 procent więcej czasu. Całkowity narzut zależy od liczby szyfrowanych kolumn oraz częstości uzyskiwania do nich dostępu, dlatego też należy szyfrować jedynie kolumny zawierające dane wrażliwe.



Więcej informacji na temat portfeli oraz zabezpieczeń bazy danych można znaleźć w podręczniku *Advanced Security Administrator's Guide* opublikowanym przez Oracle Corporation.

Kompresja danych LOB

Dane składowane w BLOB, CLOB i NCLOB mogą być kompresowane w celu zaoszczędzenia przestrzeni składowania. Na przykład poniższa instrukcja tworzy tabelę CLOB, której zawartość będzie kompresowana. Należy zauważyć użycie słowa kluczowego COMPRESS:

```
CREATE TABLE clob_content3 (
    id INTEGER PRIMARY KEY,
    clob_column CLOB
) LOB(clob_column) STORE AS SECUREFILE (
    COMPRESS
    CACHE
);
```

 Nawet jeżeli tabela nie zawiera danych szyfrowanych, konieczne jest użycie klauzuli SECUREFILE.

Przy wstawianiu danych do LOB będą one automatycznie kompresowane przez bazę danych. Przy odczytywaniu danych z LOB będą one automatycznie dekompresowane. Można również użyć opcji COMPRESS HIGH, określającej najwyższy poziom kompresji danych. Domyślnym ustawieniem jest COMPRESS MEDIUM (poziom średni), a słowo kluczowe MEDIUM jest opcjonalne. Im wyższy stopień kompresji, tym występuje wyższy narzut przy odczycie i zapisie danych LOB.

Usuwanie powtarzających się danych LOB

Typy BLOB, CLOB i NCLOB można skonfigurować tak, że wstawiane do nich powtarzające się dane będą automatycznie usuwane. Ten proces to deduplikacja, która pozwala oszczędzić przestrzeń składowania. Na przykład poniższa instrukcja tworzy tabelę z kolumną typu CLOB, której zawartość będzie deduplikowana. Należy zauważyć użycie słowa kluczowego DEDUPPLICATE:

```
CREATE TABLE clob_content2 (
    id INTEGER PRIMARY KEY,
    clob_column CLOB
) LOB(clob_column) STORE AS SECUREFILE (
    DEDUPPLICATE LOB
    CACHE
);
```

Wszystkie powtarzające się dane wstawiane do LOB będą automatycznie usuwane przez bazę danych. Do wykrywania powtarzających się danych jest stosowany bezpieczny algorytm haszujący SHA1.

Nowe właściwości dużych obiektów w Oracle Database 12c

Domyślnie w Oracle Database 12c obiekty LOB są składowane jako SecureFiles LOBs. Jest to jednoznaczne z dopisaniem opcji SECUREFILE we wcześniejszych wersjach bazy danych Oracle. Jeśli nie chcesz składać ich w ten sposób w Oracle Database 12c, należy użyć opcji BASICFILE. Tego typu obiekty LOB są nazywane *BasicFiles LOBs* i jest to oficjalna nazwa nadana przez Oracle Corporation.

Poniższa instrukcja tworzy tabelę z BASICFILE CLOB:

```
CREATE TABLE clob_content_basicfile (
    id INTEGER PRIMARY KEY,
    clob_column CLOB
) LOB(clob_column) STORE AS BASICFILE;
```

Administrator bazy danych może kontrolować tworzenie obiektów LOB za pomocą dwóch parametrów bazy danych zapisanych w pliku *init.ora*:

- **compatible** określa, z jaką wersją bazy danych należy zachować zgodność (na przykład 12.0.0.0.0 oznacza pierwszą wersję Oracle Database 12c).
- **db_securefile** określa, jak należy stosować zabezpieczone obiekty LOB (*SecureFiles LOBs*). Przykładowo opcja **PREFERRED** oznacza, że będą tworzone one domyślnie tam, gdzie będzie to możliwe.

Gdy parametr **compatible** jest ustawiony na poprawną wersję Oracle Database 12c, domyślną wartością dla **db_securefile** jest **PREFERRED**. Administrator bazy danych może ten parametr zmienić, aby zmodyfikować sposób tworzenia obiektów LOB.

Poprawne wartości **db_securefile** to:

- **PREFERRED** oznacza, że domyślnie będą tworzone zabezpieczone LOB (*SecureFiles LOBs*) tam, gdzie jest to możliwe. To nowa opcja w Oracle Database 12c.
- **PERMITTED** oznacza, że zabezpieczone LOB mogą być tworzone.
- **NEVER** zabrania tworzenia zabezpieczonych LOB. Nawet jeśli polecenie zawiera instrukcje tworzenia zabezpieczonych LOB (*SecureFiles LOBs*), zamiast nich będą tworzone niezabezpieczone LOB (*BasicFiles LOBs*). Jeżeli w poleceniu znajdzie się opcja wymuszająca kompresję, szyfrowanie lub deduplikację, baza danych zwróci błąd.
- **ALWAYS** usiłuje utworzyć zabezpieczone LOB. Obiekty LOB, które nie znajdują się w przestrzeni tabel ASSM (*Automatic Segment Space Management*), są tworzone jako *BasicFiles LOBs*, jeśli nie mają jawnie wpisanego parametru **SECUREFILE**.
- **IGNORE** uniemożliwia tworzenie zabezpieczonych LOB. Słowo kluczowe **SECUREFILE** i wszystkie związane z nim opcje są ignorowane.

Dodatkowo na sposób przechowywania LOB ma wpływ parametr **compatible**:

- Jeżeli zawiera poprawne oznaczenie wersji Oracle Database 10g, składowanie LOB odbywa się tak jak w Oracle Database 10g. Ta wersja bazy danych umożliwia przechowywanie jedynie niezabezpieczonych LOB (*BasicFiles LOBs*), a słowa kluczowe **BASICFILE** i **SECUREFILE** nie są obsługiwane.
- Jeżeli zawiera poprawne oznaczenie wersji Oracle Database 11g, możliwe jest używanie LOB zabezpieczonych i niezabezpieczonych. Jeśli parametr ten nie jest ustawiony na wersję 11.1 lub wyższą, obiekty LOB nie są traktowane jako zabezpieczone LOB (*SecureFiles LOBs*). Są one traktowane jak niezabezpieczone LOB (*BasicFiles LOBs*).
- Jeżeli zawiera poprawne oznaczenie wersji Oracle Database 12c, należy jawnie użyć opcji **BASICFILE**, by uzyskać niezabezpieczony LOB. W przeciwnym przypadku polecenie **CREATE TABLE** utworzy domyślnie zabezpieczony LOB.

Jeszcze więcej informacji na temat dużych obiektów można znaleźć w podręczniku *Oracle Database Large Objects Developer's Guide* opublikowanym przez Oracle Corporation.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- obiekty LOB mogą być używane do składowania danych binarnych, danych znakowych oraz odwołań do plików zewnętrznych,
- w obiekcie LOB można przechować maksymalnie 128 terabajtów danych,
- istnieją cztery typy dużych obiektów: **CLOB**, **NCLOB**, **BLOB** i **BFILE**,
- w **CLOB** składowane są dane znakowe,
- w **NCLOB** składowane są wielobajtowe dane znakowe,
- w **BLOB** składowane są dane binarne,
- w **BFILE** składowane są wskaźniki do plików znajdujących się w systemie plików,
- pakiet PL/SQL **DBMS_LOB** zawiera metody umożliwiające pracę z dużymi obiektami.

W kolejnym rozdziale nauczysz się, jak optymalizować instrukcje SQL.

ROZDZIAŁ

16

Optymizacja SQL

W tym rozdziale:

- dowiesz się czegoś o optymalizacji instrukcji SQL,
- poznasz wskazówki pozwalające skrócić czas wykonywania zapytań,
- zapoznasz się z informacjami na temat optymalizatora Oracle,
- dowiesz się, jak porównać koszt wykonywania zapytań,
- przyjrzymy się podpowiedziom optymalizatora,
- poznasz inne narzędzia optymalizujące.

Podstawowe informacje o optymalizacji SQL

Jedna z głównych zalet SQL jest związana z tym, że nie musimy dokładnie przekazywać bazie danych, jak ma uzyskać żądane dane. Po prostu uruchamiamy zapytanie, określając żądaną informację, a oprogramowanie bazy danych opracowuje najlepszy sposób ich pobrania.

Czasami można zwiększyć wydajność instrukcji SQL przez ich optymalizację. W kolejnych podrozdziałach znajduje się kilka porad dotyczących optymalizacji, które mogą przyspieszyć wykonywanie zapytań. Później zajmiemy się bardziej zaawansowanymi technikami optymalizacji. Przykłady z tego rozdziału korzystają ze schematu `store`.

Należy filtrować wiersze za pomocą klauzuli WHERE

Wielu nowicjuszy pobiera z tabeli wszystkie wiersze, choć potrzebują tylko jednego (lub kilku). To jest straszne marnotrawstwo. Lepsze rozwiązanie polega na dodaniu do zapytania klauzuli `WHERE`. Dzięki temu ograniczamy pobieranie wierszy jedynie do tych, których potrzebujemy.

Załóżmy, że chcemy uzyskać informacje na temat klientów 1 i 2. Poniższe zapytanie pobiera wszystkie wiersze z tabeli `customers` w schemacie `store` (marnotrawstwo):

```
-- ŹLE (pobiera wszystkie wiersze z tabeli customers)
SELECT *
FROM customers;

CUSTOMER_ID FIRST_NAME LAST_NAME   DOB      PHONE
-----  -----
1 Jan        Nikiel      65/01/01 800-555-1211
2 Lidia      Stal       68/02/05 800-555-1212
```

3 Stefan	Brąz	71/03/16	800-555-1213
4 Grażyna	Cynk		800-555-1214
5 Jadwiga	Mosiądz	70/05/20	

W kolejnym zapytaniu dodano klauzulę WHERE w celu pobrania jedynie informacji o klientach 1 i 2:

-- DOBRZE (użyto klauzuli WHERE ograniczającej pobierane wiersze)

```
SELECT *
FROM customers
WHERE customer_id IN (1, 2);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jan	Nikiel	65/01/01	800-555-1211
2	Lidia	Stal	68/02/05	800-555-1212

Należy unikać stosowania funkcji w klauzuli WHERE, ponieważ wydłuża to czas wykonywania polecenia.

Należy używać złączeń tabel zamiast wielu zapytań

Jeżeli potrzebne są informacje z kilku związkanych z sobą tabel, należy użyć warunków złączenia zamiast wielu zapytań. W poniższym przykładzie złego postępowania użyto dwóch zapytań do pobrania nazwy produktu oraz nazwy typu produktu dla produktu nr 1 (użycie dwóch zapytań to marnotrawstwo). Pierwsze zapytanie pobiera wartości kolumny name i product_type_id z tabeli products dla produktu nr 1. Drugie zapytanie używa tej wartości product_type_id do pobrania wartości kolumny name z tabeli product_types:

-- ŹLE (dwa osobne zapytania, a można użyć jednego)

```
SELECT name, product_type_id
FROM products
WHERE product_id = 1;
```

NAME	PRODUCT_TYPE_ID
Nauka współczesna	1

```
SELECT name
FROM product_types
WHERE product_type_id = 1;
```

NAME

Książka

Zamiast dwóch zapytań należy napisać jedno wykorzystujące złączenie między tabelami products i product_types. Obrazuje to poniższy przykład:

-- DOBRZE (jedno zapytanie ze złączeniem)

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME
Nauka współczesna	Książka

W wyniku tego zapytania pobierana jest ta sama nazwa produktu i ta sama nazwa typu produktu co w pierwszym przykładzie, wyniki są jednak uzyskiwane za pomocą jednego zapytania. Jedno zapytanie jest najczęściej bardziej wydajne niż dwa.

Kolejność złączenia w zapytaniu należy dobrać tak, aby kolejne złączane tabele zawierały coraz mniejszą liczbę wierszy. Założymy, że łączymy trzy powiązane tabele tab1, tab2 i tab3. tab1 zawiera 1000 wierszy, tab2 ma 100 wierszy, a tab3 — 10 wierszy. Należy najpierw złączyć tab1 i tab2, a następnie tab2 i tab3.

Należy unikać złożonych perspektyw w zapytaniach, ponieważ w takim przypadku najpierw są wykonywane zapytania dla widoków, a następnie zasadnicze zapytanie. Zamiast tego należy pisać zapytania z użyciem tabel, a nie perspektyw.

Wykonując związania, należy używać w pełni kwalifikowanych odwołań do kolumn

W zapytaniach należy zawsze umieszczać aliasy tabel i używać aliasu dla każdej kolumny w zapytaniu (nazywamy to pełnym kwalifikowaniem odwołań do kolumn). Dzięki temu baza danych nie musi wyszukiwać każdej kolumny w tabelach używanych w zapytaniu.

Poniższy przykład złego rozwiązania używa aliasów `p` i `pt` dla tabel `products` i `product_type` (odpowiednio), ale zapytanie nie w pełni kwalifikuje kolumny `description` i `price`:

```
-- ŹLE (kolumny description i price nie są w pełni kwalifikowane)
SELECT p.name, pt.name, description, price
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME	
DESCRIPTION		PRICE
Nauka współczesna Opis współczesnej nauki	Książka	19,95

Ten przykład działa, ale baza danych musi wyszukać kolumny `description` i `price` w obu tabelach. Dzieje się tak, ponieważ żaden alias nie informuje, w której tabeli znajdują się te kolumny. Dodatkowy czas, który jest przeznaczony na wyszukiwanie, jest czasem zmarnowanym.

Poniższy przykład dobrej praktyki zawiera alias tabeli `p` w celu pełnej kwalifikacji kolumn `description` i `price`:

```
-- DOBRZE (wszystkie kolumny są w pełni kwalifikowane)
SELECT p.name, pt.name, p.description, p.price
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME	
DESCRIPTION		PRICE
Nauka współczesna Opis współczesnej nauki	Książka	19,95

Ponieważ wszystkie odwołania do kolumn zawierają alias tabeli, baza danych nie traci czasu na wyszukiwanie kolumn w tabelach, czas wykonywania jest więc krótszy.

Należy używać wyrażeń CASE zamiast wielu zapytań

Jeżeli konieczne jest przeprowadzenie wielu obliczeń z użyciem tych samych wierszy tabeli, należy zastosować wyrażenia CASE zamiast wielu zapytań. W poniższym przykładzie złego rozwiązania użyto kilku zapytań do policzenia produktów w różnych przedziałach cenowych:

```
-- ŹLE (trzy osobne zapytania tam, gdzie wystarczy jedna instrukcja CASE)
SELECT COUNT(*)
```

```

FROM products
WHERE price < 13;

COUNT(*)
-----
2

SELECT COUNT(*)
FROM products
WHERE price BETWEEN 13 AND 15;

COUNT(*)
-----
5

SELECT COUNT(*)
FROM products
WHERE price > 15;

COUNT(*)
-----
5

```

Zamiast używać trzech zapytań, wystarczy napisać jedno zapytanie wykorzystujące wyrażenie CASE. Obrazuje to poniższy przykład:

```
-- DOBRZE (jedno zapytanie z wyrażeniem CASE)
SELECT
  COUNT(CASE WHEN price < 13 THEN 1 ELSE null END) niska,
  COUNT(CASE WHEN price BETWEEN 13 AND 15 THEN 1 ELSE null END) średnia,
  COUNT(CASE WHEN price > 15 THEN 1 ELSE null END) wysoka
FROM products;
```

NISKIE	ŚREDNIE	WYSOKIE
2	5	5

Liczba produktów z ceną niższą niż 13 zł jest oznaczona jako **niska**, produkty których cena mieści się w przedziale 13 – 15 zł są oznaczone etykietą **średnia**, a produkty o cenie wyższej niż 15 zł są oznaczone jako **wysoka**.



Oczywiście, w wyrażeniach CASE można stosować nakładające się przedziały, a także różnego rodzaju funkcje.

Należy dodać indeksy do tabel

Szukając konkretnych informacji w książce, możemy albo przeszukać całą, albo skorzystać z indeksu, aby zlokalizować odpowiedni fragment. Indeks tabeli bazy danych pełni podobną funkcję co indeks w książce, jest on jednak używany do wyszukiwania konkretnych wierszy w tabeli. Wadą indeksów jest to, że przy wstawianiu wiersza do tabeli jest wymagany dodatkowy czas potrzebny na aktualizację indeksu.

Zazwyczaj za tworzenie indeksów odpowiedzialny jest administrator bazy danych, jednak jako twórcy aplikacji powinieneś dostarczyć administratorowi wskazówek, które kolumny powinny być indeksowane, ponieważ lepiej znasz działanie aplikacji.

W rozdziale 11. opisane zostały dwa rodzaje indeksów:

- indeksy typu B-drzewo,
- indeksy bitmapowe.

Kolejne podrozdziały zawierają podsumowanie zaleceń, kiedy używać tego typu indeksów. W rozdziale 11. indeksy zostały opisane bardziej dokładnie.

Kiedy tworzyć indeks typu B-drzewo

Ogólnie rzecz biorąc, indeks typu B-drzewo powinien być tworzony, jeżeli pobierana jest niewielka liczba wierszy z tabeli zawierającej wiele wierszy. Standardowa zasada brzmi:

Twórz indeks typu B-drzewo, jeżeli zapytanie pobiera <= 10 procent całkowitej liczby wierszy w tabeli.

To oznacza, że kolumna indeksu typu B-drzewo powinna zawierać szeroki zakres wartości. Dobrym pomysłem na indeks jest kolumna zawierająca unikatową wartość dla każdego wiersza (na przykład numer PESEL); kiepskim materiałem do indeksowania jest kolumna zawierająca niewielki zakres wartości (na przykład N, S, E i W lub 1, 2, 3, 4, 5, 6). Baza danych Oracle automatycznie tworzy indeks typu B-drzewo na kluczu głównym tabeli oraz kolumnach z ograniczeniem unikatowości.

Jeżeli dane z bazy są często pobierane za pomocą zapytań hierarchicznych (zawierających klauzule CONNECT BY), należy dodać indeksy typu B-drzewo do kolumn, do których występują odwołania w klauzulach START WITH i CONNECT BY (informacje na temat zapytań hierarchicznych znajdują się w rozdziale 7.).

Kiedy tworzyć indeks bitmapowy

Jeżeli kolumna zawiera niewielki zakres wartości i jest często używana w klauzuli WHERE, można rozważyć dodanie do niej indeksu bitmapowego. Indeksy bitmapowe są zwykle używane w hurtowniach danych, które są bazami danych zawierającymi ogromne ilości informacji. Dane z hurtowni danych są zwykle odczytywane z użyciem wielu zapytań, ale nie są one często modyfikowane przez wiele współbieżnych transakcji.

Należy stosować klauzulę WHERE zamiast HAVING

Klauzula WHERE służy do filtrowania wierszy, klauzula HAVING — do filtrowania grup wierszy. Ponieważ klauzula HAVING filtryuje grupy wierszy po tym, jak zostały pogrupowane (co zajmuje nieco czasu), jeżeli tylko to możliwe, należy najpierw przefiltrować wiersze za pomocą klauzuli WHERE. Dzięki temu można oszczędzić czas potrzebny na pogrupowanie przefiltrowanych wierszy.

Poniższe niepoprawne zapytanie pobiera product_type_id i średnią cenę produktów, których product_type_id wynosi 1 lub 2. To zapytanie wykonuje następujące czynności:

- Używa klauzuli GROUP BY do pogrupowania wierszy w bloki o tej samej wartości product_type_id.
- Używa klauzuli HAVING do przefiltrowania zwróconych wyników według wartości product_type_id (1 lub 2). Nie jest to dobre rozwiązanie, ponieważ wystarczyłaby klauzula WHERE.

```
-- ŹLE (użyto HAVING zamiast WHERE)
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING product_type_id IN (1, 2);

PRODUCT_TYPE_ID AVG(PRICE)
-----
1      24,975
2      26,22
```

W poniższym prawidłowym zapytaniu zastosowano klauzulę WHERE zamiast HAVING, aby najpierw wybrać wiersze, których product_type_id ma wartość 1 lub 2:

```
-- DOBRZE (użyto WHERE zamiast HAVING)
SELECT product_type_id, AVG(price)
FROM products
WHERE product_type_id IN (1, 2)
GROUP BY product_type_id;

PRODUCT_TYPE_ID AVG(PRICE)
-----
1      24,975
2      26,22
```

Należy używać UNION ALL zamiast UNION

UNION ALL pobiera wszystkie wiersze zwrócone przez dwa zapytania, włącznie z duplikatami, natomiast UNION zwraca wszystkie niepowtarzające się wiersze pobrane przez zapytania. Ponieważ UNION usuwa duplikaty (co zajmuje trochę czasu), jeżeli tylko jest to możliwe, należy stosować UNION ALL.

W poniższym nieprawidłowym zapytaniu użyto UNION (jest to rozwiązywanie złe, ponieważ wystarczy zastosowanie UNION ALL) do pobrania wierszy z tabel products i more_products. Należy zwrócić uwagę, że pobrane zostały wszystkie niepowtarzające się wiersze z tabel products i more_products:

```
-- ŹLE (użyto UNION zamiast UNION ALL)
SELECT product_id, product_type_id, name
FROM products
UNION
SELECT prd_id, prd_type_id, name
FROM more_products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Chemia
3	2	Supernowa
3		Supernowa
4	2	Lądowanie na Księżycu
4	2	Wojny czołgów
5	2	Z Files
5	2	Łódź podwodna
6	2	2412: Powrót
7	3	Space Force 9
8	3	Z innej planety
9	4	Muzyka klasyczna
10	4	Pop 3
11	4	Twórczy wrzask
12		Pierwsza linia

W poniższym dobrym rozwiązyaniu zmieniono wcześniejsze zapytanie tak, aby zawierało UNION ALL. Należy zauważać, że zostały zwrócone wszystkie wiersze z tabel products i more_products, łącznie z duplikatami:

```
-- DOBRZE (użyto UNION ALL zamiast UNION)
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Nauka współczesna
2	1	Chemia
3	2	Supernowa
4	2	Wojny czołgów
5	2	Z Files
6	2	2412: Powrót
7	3	Space Force 9
8	3	Z innej planety
9	4	Muzyka klasyczna
10	4	Pop 3
11	4	Twórczy wrzask
12		Pierwsza linia
1	1	Nauka współczesna
2	1	Chemia
3		Supernowa
4	2	Lądowanie na Księżycu
5	2	Łódź podwodna

Należy używać EXISTS zamiast IN

Operator IN służy do sprawdzenia, czy wartość znajduje się na liście. Operator EXISTS sprawdza, czy podzapytanie zwraca jakieś wiersze. Operator EXISTS różni się od IN, ponieważ sprawdza istnienie wierszy, natomiast IN sprawdza faktyczne wartości. W podzapytaniach operator EXISTS zwykle oferuje większą wydajność niż IN, dlatego jeżeli jest to możliwe, należy go stosować zamiast IN.

Szczegółowe informacje na temat zastosowania operatora EXISTS w podzapytaniach skorelowanych znajdują się w podrozdziale „Użycie operatorów EXISTS i NOT EXISTS z podzapytaniem skorelowanym” w rozdziale 6. Należy pamiętać, że zapytania skorelowane obsługują wartości NULL.

W poniższym przykładzie złego zapytania użyto operatora IN (jest to rozwiązanie złe, ponieważ wystarczyłyby operator EXISTS) do pobrania produktów, które zostały kupione:

```
-- ŹLE (użyto IN zamiast EXISTS)
SELECT product_id, name
FROM products
WHERE product_id IN
  (SELECT product_id
   FROM purchases);
```

PRODUCT_ID	NAME
1	Nauka współczesna
2	Chemia
3	Supernowa

W poniższym przykładzie zmieniono wcześniejsze zapytanie tak, aby wykorzystywało operator EXISTS:

```
-- DOBRZE (użyto EXISTS, a nie IN)
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);
```

PRODUCT_ID	NAME
1	Nauka współczesna
2	Chemia
3	Supernowa

Należy używać EXISTS zamiast DISTINCT

Możemy powstrzymać wyświetlanie duplikatów za pomocą opcji DISTINCT. Operator EXISTS sprawdza, czy podzapytanie zwraca jakieś wiersze. Jeżeli tylko jest to możliwe, należy używać EXISTS zamiast DISTINCT, ponieważ DISTINCT sortuje pobrane wiersze przed odrzuceniem powtarzających się.

W poniższym przykładzie złego zapytania użyto DISTINCT do pobrania produktów, które zostały kupione (wystarczyłyby operator EXISTS):

```
-- ŹLE (użyto DISTINCT tam, gdzie wystarczy EXISTS)
SELECT DISTINCT pr.product_id, pr.name
FROM products pr, purchases pu
WHERE pr.product_id = pu.product_id;
```

PRODUCT_ID	NAME
1	Nauka współczesna
2	Chemia
3	Supernowa

W poniższym prawidłowym zapytaniu użyto operatora EXISTS zamiast opcji DISTINCT:

```
-- DOBRZE (użyto EXISTS zamiast DISTINCT)
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);

PRODUCT_ID NAME
-----
1 Nauka współczesna
2 Chemia
3 Supernowa
```

Należy używać GROUPING SETS zamiast CUBE

Klauzula GROUPING SETS zwykle jest bardziej wydajna niż CUBE. Dlatego też, jeżeli jest to możliwe, należy używać jej zamiast CUBE. Ten temat został dokładnie opisany w podrozdziale „Klauzula GROUPING SETS” w rozdziale 7.

Należy stosować zmienne dowiązane

Oprogramowanie bazy danych zachowuje instrukcje SQL w pamięci podręcznej. Buforowana instrukcja SQL jest używana ponownie, jeżeli do bazy danych zostanie przesłana identyczna. Instrukcja SQL musi być jednak *zupełnie identyczna*, aby mogła zostać użyta ponownie. To oznacza, że:

- wszystkie znaki w instrukcji SQL muszą być takie same,
- wszystkie litery w SQL muszą mieć taką samą wielkość,
- wszystkie odstępy w instrukcji SQL muszą być takie same.

Jeżeli chcemy przesyłać inne wartości kolumn w instrukcji, możemy użyć zmiennych dowiązanych zamiast literałowych wartości kolumn. Te założenia zostaną przedstawione na przykładach.

Nieidentyczne instrukcje SQL

W tym podrozdziale zostaną przedstawione nieidentyczne instrukcje SQL. Poniższe nieidentyczne zapytania pobierają produkt nr 1 i 2:

```
SELECT * FROM products WHERE product_id = 1;
SELECT * FROM products WHERE product_id = 2;
```

Te zapytania nie są identyczne, ponieważ w pierwszym użyto wartości 1, a w drugim wartości 2.

W poniższych nieidentycznych zapytaniach odstępy znajdują się w różnych miejscach:

```
SELECT * FROM products WHERE product_id = 1;
SELECT * FROM products WHERE product_id = 1;
```

Poniższe nieidentyczne zapytania różnią się wielkością niektórych liter:

```
select * from products where product_id = 1;
SELECT * FROM products WHERE product_id = 1;
```

Gdy zobaczyliśmy już kilka nieidentycznych instrukcji, zerknijmy na identyczne instrukcje SQL korzystające ze zmiennych dowiązanych.

Identyczne instrukcje SQL korzystające ze zmiennych dowiązanych

Możemy zapewnić identyczność instrukcji dzięki użyciu zmiennych dowiązanych do reprezentowania wartości kolumn. Zmienną dowiązaną tworzymy za pomocą polecenia SQL*Plus VARIABLE.

Na przykład poniższe polecenie tworzy zmienną typu NUMBER o nazwie `v_product_id`:

VARIABLE v_product_id NUMBER

 **Uwaga** Do zdefiniowania typu zmiennej dowiązanej można użyć każdego z typów wymienionych w tabeli A.1 w dodatku A.

Odwzorowanie do zmiennej dowiązanej w instrukcji SQL lub PL/SQL wykonujemy za pomocą dwukropka i nazwy zmiennej (na przykład `:v_product_id`). Poniższy blok PL/SQL przypisuje zmiennej `v_product_id` wartość 1:

```
BEGIN
  :v_product_id := 1;
END;
/
```

W poniższym zapytaniu użyto zmiennej `v_product_id` do ustawienia wartości kolumny `product_id` w klauzuli WHERE. Ponieważ w prezentowanym bloku PL/SQL zmiennej `v_product_id` przypisano wartość 1, zapytanie pobiera informacje o produkcie nr 1:

```
SELECT * FROM products WHERE product_id = :v_product_id;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	DESCRIPTION	PRICE
1	1	Nauka współczesna	Opis współczesnej nauki	19,95

W kolejnym przykładzie przypisano zmiennej `v_product_id` wartość 2 i powtórzono zapytanie:

```
BEGIN
  :v_product_id := 2;
END;
/
SELECT * FROM products WHERE product_id = :v_product_id;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	DESCRIPTION	PRICE
2	1	Chemia	Wprowadzenie do chemii	30

Ponieważ zapytanie użyte w tym przykładzie jest takie samo jak to użyte w poprzednim przykładzie, ponownie zostaje wykorzystane zapytanie przechowywane w pamięci podręcznej, co powoduje poprawę wydajności.

 **Wskazówka** Jeżeli często jest wykonywane to samo zapytanie, należy stosować zmienne dowiązane. W przykładzie są one specyficzne dla sesji i należy je ponownie ustawić po przerwaniu jej.

Wypisywanie listy i wartości zmiennych dowiązanych

W SQL*Plus listę zmiennych wyświetla się za pomocą polecenia VARIABLE. Na przykład:

VARIABLE
variable v_product_id
datatype NUMBER

Do wyświetlania wartości zmiennej dowiązanej w SQL*Plus służy polecenie PRINT. Na przykład:

```
PRINT v_product_id
V_PRODUCT_ID
-----
2
```

Użycie zmiennej dowiązanej do składowania wartości zwróconej przez funkcję PL/SQL

Zmienna dowiązana może również zostać użyta do składowania wartości zwróconych z funkcji PL/SQL. W poniższym przykładzie jest tworzona zmienna dowiązana o nazwie `v_average_product_price`, w której zapisywana jest wartość zwrócona przez funkcję `average_product_price()` (ta funkcja została opisana w rozdziale 12. i oblicza średnią cenę produktów z przesłaną wartością `product_type_id`):

```
VARIABLE v_average_product_price NUMBER
BEGIN
  :v_average_product_price := average_product_price(1);
END;
/
PRINT v_average_product_price
```

V_AVERAGE_PRODUCT_PRICE

24,975

Użycie zmiennej dowiązanej do składowania wierszy z REFCURSOR

W zmiennej dowiązanej można składować wartości zwrócone z REFCURSOR (jest to wskaźnik do listy wierszy). W poniższym przykładzie jest tworzona zmienna o nazwie `v_products_refcursor`, do której zapisywane są wyniki zwrócone przez funkcję `product_package.get_products_ref_cursor()`. Ta funkcja została wprowadzona w rozdziale 12. i zwraca wskaźnik do wierszy w tabeli `products`:

```
VARIABLE v_products_refcursor REFCURSOR
BEGIN
  :v_products_refcursor := product_package.get_products_ref_cursor();
END;
/
PRINT v_products_refcursor
```

PRODUCT_ID	NAME	PRICE
1	Nauka współczesna	19,95
2	Chemia	30
3	Supernowa	25,99
4	Wojny czołgów	13,95
5	Z Files	49,99
6	2412: Powrót	14,95
7	Space Force 9	13,49
8	Z innej planety	12,99
9	Muzyka klasyczna	10,99
10	Pop 3	15,99
11	Twórczy wrzask	14,99
12	Pierwsza linia	13,49

Porównywanie kosztu wykonania zapytań

W oprogramowaniu bazy danych wykorzystywany jest podsystem zwany **optymalizatorem**, który generuje najbardziej wydajną ścieżkę dostępu do danych składowanych w tabelach. Ścieżkę wygenerowaną przez optymalizator nazywamy **planem wykonania**. W Oracle Database 10g i nowszych statystyki danych w tabelach i indeksach są zbierane automatycznie w celu wygenerowania najlepszego planu wykonania (nazywamy to **optymalizacją w oparciu o koszt**).

Porównywanie planów wykonania generowanych przez optymalizator pozwala nam ocenić wzajemny koszt wykonania różnych instrukcji SQL. Wyniki możemy wykorzystać do poprawienia instrukcji SQL. W tym podrozdziale dowiesz się, jak przeglądać i interpretować kilka planów wykonania.



W wersjach wcześniejszych niż Oracle Database 10g statystyki nie są zbierane automatycznie, a optymalizator automatycznie wybiera domyślną optymalizację opartą na regułach. W optymalizacji opartej na regułach do utworzenia planu wykonywania są wykorzystywane reguły syntaktyczne. Optymalizacja oparta na koszcie jest zwykle lepsza niż oparta na regułach, ponieważ wykorzystuje faktyczne informacje zebrane z danych w tabelach i indeksach. Jeżeli używana jest baza Oracle Database 9i lub wcześniejsza, można samodzielnie zbierać statystyki. Więcej informacji na ten temat znajduje się w podrozdziale „Zbieranie statystyk tabeli”.

Przeglądanie planów wykonania

Optymalizator generuje plan wykonywania dla instrukcji SQL. Można go przejrzeć za pomocą polecenia EXPLAIN PLAN w SQL*Plus. Umieszcza ono plan wykonania instrukcji w tabeli o nazwie `plan_table` (tę tabelę często nazywamy **tabelą planu**). W celu przejrzenia tego planu wykonania należy odpytać tabelę planu. Najpierw jednak należy upewnić się, że aktualnie istnieje ona w bazie danych.

Sprawdzanie, czy tabela planu aktualnie istnieje w bazie danych

Aby sprawdzić, czy tabela planu istnieje w bazie danych, należy połączyć się z bazą jako użytkownik `store` i uruchomić następujące polecenie DESCRIBE:

```
SQL> DESCRIBE plan_table
```

Name	NULL?	Type
STATEMENT_ID		VARCHAR2(30)
PLAN_ID		NUMBER
TIMESTAMP		DATE
REMARKS		VARCHAR2(4000)
OPERATION		VARCHAR2(30)
OPTIONS		VARCHAR2(255)
OBJECT_NODE		VARCHAR2(128)
OBJECT_OWNER		VARCHAR2(128)
OBJECT_NAME		VARCHAR2(128)
OBJECT_ALIAS		VARCHAR2(261)
OBJECT_INSTANCE		NUMBER(38)
OBJECT_TYPE		VARCHAR2(30)
OPTIMIZER		VARCHAR2(255)
SEARCH_COLUMNS		NUMBER
ID		NUMBER(38)
PARENT_ID		NUMBER(38)
DEPTH		NUMBER(38)
POSITION		NUMBER(38)
COST		NUMBER(38)
CARDINALITY		NUMBER(38)
BYTES		NUMBER(38)
OTHER_TAG		VARCHAR2(255)
PARTITION_START		VARCHAR2(255)
PARTITION_STOP		VARCHAR2(255)
PARTITION_ID		NUMBER(38)
OTHER		LONG
OTHER_XML		CLOB
DISTRIBUTION		VARCHAR2(30)
CPU_COST		NUMBER(38)
IO_COST		NUMBER(38)
TEMP_SPACE		NUMBER(38)
ACCESS_PREDICATES		VARCHAR2(4000)
FILTER_PREDICATES		VARCHAR2(4000)
PROJECTION		VARCHAR2(4000)
TIME		NUMBER(38)
QBLOCK_NAME		VARCHAR2(30)

Jeżeli zostanie wyświetlony opis tabeli podobny do powyższego, tabela planu już istnieje. Jeżeli zostanie wyświetlony komunikat o błędzie, należy utworzyć tabelę planu.

Tworzenie tabeli planu

Jeżeli tabela planu nie istnieje, należy ją utworzyć. W tym celu należy uruchomić skrypt SQL*Plus o nazwie *utlxplan.sql*. W systemie Windows ten skrypt może znajdować się w katalogu *C:\oracle12c\product\12.1.0\dbhome_1\RDBMS\ADMIN*. Na komputerze z systemem Linux skrypt może znajdować się w katalogu */u01/app/oracle12c/product/12.1.0/dbhome_1/rdbms/admin*.

Poniżej przedstawiono polecenie uruchamiające skrypt *utlxplan.sql*:

```
SQL> @ C:\oracle12c\product\12.1.0\dbhome_1\RDBMS\ADMIN\utlxplan.sql
```



Należy użyć ścieżki dostępu zgodnej z własną konfiguracją.

Najważniejsze kolumny tabeli planu zostały opisane w tabeli 16.1.

Tabela 16.1. Kolumny tabeli planu

Kolumna	Opis
statement_id	Nazwa, którą przypisaliśmy planowi wykonania
operation	Wykonywana operacja bazy danych. Może to być: <ul style="list-style-type: none"> ■ skanowanie tabeli ■ skanowanie indeksu ■ sięganie po wiersze z tabeli z użyciem indeksu ■ tworzeniełączenia dwóch tabel ■ sortowanie zestawu wierszy Na przykład operacja uzyskiwania dostępu do tabeli to TABLE ACCESS
options	Nazwa opcji użytej w operacji. Na przykład opcją dla pełnego skanowania jest FULL
object_name	Nazwa obiektu bazy danych używanego przez operację
object_type	Atrybut obiektu. Na przykład indeks unikatowy ma atrybut UNIQUE
id	Liczba przypisana danej operacji w planie wykonania
parent_id	Liczba nadzędna dla bieżącego kroku planu wykonania. Wartość parent_id jest związana z wartością id w kroku nadzędnym
position	Kolejność przetwarzania kroków z tym samym parent_id
cost	Szacowane jednostki pracy dla operacji. W przypadku optymalizacji opartej na koszcie jako jednostki pracy używane są liczba operacji We/Wy dysku, wykorzystanie CPU i wykorzystanie pamięci. Dlatego też koszt jest wypadkową liczbą operacji dyskowych oraz ilości czasu procesora i pamięci użytych w wykonywaniu operacji

Tworzenie centralnej tabeli planu

Jeżeli jest to konieczne, administrator bazy danych może utworzyć jedną, centralną tabelę planu. Dzięki temu poszczególni użytkownicy nie będą musieli tworzyć własnych tabel planu. W tym celu administrator musi wykonać następujące kroki:

1. utworzyć tabelę planu w wybranym schemacie przez uruchomienie skryptu *utlxplan.sql*,
2. utworzyć publiczny synonim dla tabeli planu,
3. przydzielić roli publicznej uprawnienia dostępu do tabeli planu.

Oto przykład:

```
@ E:\oracle12c\product\12.1.0\dbhome_1\RDBMS\ADMIN\utlxplan.sql
CREATE PUBLIC SYNONYM plan_table FOR plan_table;
GRANT SELECT, INSERT, UPDATE, DELETE ON plan_table TO PUBLIC;
```

Generowanie planu wykonania

Po utworzeniu tabeli planu możemy użyć polecenia EXPLAIN PLAN do wygenerowania planu wykonania instrukcji SQL. Składnia polecenia EXPLAIN PLAN ma następującą postać:

```
EXPLAIN PLAN SET STATEMENT_ID = statement_id FOR instrukcja_sql;
```

gdzie:

- *statement_id* jest nazwą, którą chcemy nadać planowi wykonania. Może to być dowolny tekst alfanumeryczny.
- *instrukcja_sql* jest instrukcją SQL, dla której chcemy wygenerować plan wykonania.

Poniższy przykład generuje plan wykonania dla zapytania pobierającego wszystkie wiersze z tabeli *customers* (należy zwrócić uwagę, że *statement_id* ustawiono na 'CUSTOMERS'):

```
EXPLAIN PLAN SET STATEMENT_ID = 'CUSTOMERS' FOR
SELECT customer_id, first_name, last_name FROM customers;
Explained.
```

Po wykonaniu polecenia możemy przejrzeć plan wykonania zapisany do tabeli planu. Za chwilę zobaczysz, jak to zrobić.



Zapytanie w instrukcji EXPLAIN PLAN nie zwraca wierszy z tabeli *customers*. Instrukcja ta po prostu generuje plan wykonania, który zostałby użyty przy wykonaniu tego zapytania.

Odpitywanie tabeli planu

W katalogu SQL znajduje się skrypt SQL*Plus o nazwie *explain_plan.sql*, odpytujący tabelę planu. Skrypt просi o wpisanie wartości *statement_id*, a następnie wyświetla plan wykonania tej instrukcji.

Skrypt *explain_plan.sql* zawiera następujący kod:

```
-- wyświetla plan wykonania dla określonego statement_id

UNDEFINE v_statement_id;

SELECT
  id ||
  DECODE(id, 0, '', LPAD(' ', 2*(level - 1))) || ' ' ||
  operation ||
  options ||
  object_name ||
  object_type ||
  DECODE(cost, NULL, '', 'Koszt = ' || position)
AS execution_plan
FROM plan_table
CONNECT BY PRIOR id = parent_id
AND statement_id = '&&v_statement_id'
START WITH id = 0
AND statement_id = '&v_statement_id';
```

Plan wykonania tworzy hierarchię operacji bazy danych, która przypomina drzewo. Szczegóły tych operacji są składowane w tabeli planu. Operacja o id równym 0 jest korzeniem hierarchii, a pozostałe operacje wyrastają z niego. Zapytanie w skrypcie pobiera informacje o operacjach, rozpoczynając od głównej i przechodząc w głąb drzewa.

Poniższy przykład pokazuje, jak uruchomić skrypt *explain_plan.sql* w celu pobrania wcześniej utworzonego planu 'CUSTOMERS':

```
SQL> @ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: CUSTOMERS
old 12: AND statement_id = '&&v_statement_id'
new 12: AND statement_id = 'CUSTOMERS'
old 14: AND statement_id = '&v_statement_id'
new 14: AND statement_id = 'CUSTOMERS'
```

EXECUTION_PLAN

```
0 SELECT STATEMENT Koszt = 3
1   TABLE ACCESS FULL CUSTOMERS TABLE Koszt = 1
```

Operacje przedstawione w kolumnie EXECUTION_PLAN są wykonywane w następującej kolejności:

- najpierw jest wykonywana operacja najbardziej wcięta do prawej, a następnie operacje nadrzędne znajdujące się nad nią,
- w przypadku operacji o takim samym wcięciu najpierw jest wykonywana operacja wymieniona wyżej oraz wszystkie znajdujące się nad nią operacje nadrzędne.

Każda operacja zwraca wyniki w góre łańcucha, do bezpośredniej operacji nadrzędnej, po czym jest wykonywana ta operacja nadrzędna. W kolumnie EXECUTION_PLAN identyfikator operacji jest wyświetlany po lewej stronie. W przykładowym planie wykonania najpierw jest wykonywana instrukcja 1, a jej wyniki są przesyłane do operacji 0.

Poniższy przykład obrazuje kolejkowanie bardziej złożonego przykładu:

```
0 SELECT STATEMENT Koszt = 6
1   MERGE JOIN Koszt = 1
2     TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Koszt = 1
3       INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Koszt = 1
4     SORT JOIN Koszt = 2
5       TABLE ACCESS FULL PRODUCTS TABLE Koszt = 1
```

W tym przykładzie operacje będą wykonywane w kolejności 3, 2, 5, 4, 1 i 0.

Gdy znamy już kolejność wykonywania operacji, dowiedzmy się, co rzeczywiście wykonują operacje. Plan wykonania zapytania 'CUSTOMERS' był taki:

```
0 SELECT STATEMENT Koszt = 3
1   TABLE ACCESS FULL CUSTOMERS TABLE Koszt = 1
```

Na początku jest wykonywana operacja 1, której wyniki są przesyłane do operacji 0. Operacja 1 wiąże się z pełnym skanowaniem (na co wskazuje napis TABLE ACCESS FULL) tabeli customers. Oto pierwotne polecenie użyte do wygenerowania zapytania 'CUSTOMERS':

```
EXPLAIN PLAN SET STATEMENT_ID = 'CUSTOMERS' FOR
SELECT customer_id, first_name, last_name FROM customers;
```

Pełne skanowanie tabeli jest wykonywane, ponieważ instrukcja SELECT określa, że z tabeli customers mają zostać pobrane wszystkie wiersze.

Całkowity koszt zapytania wynosi 3 jednostki pracy, na co wskazuje część z kosztem znajdująca się po prawej stronie operacji w planie wykonania (0 SELECT STATEMENT Koszt = 3). Jednostka pracy jest miarą ilości przetwarzania, które musi wykonać oprogramowanie bazy danych w celu wykonania danej operacji. Im wyższy koszt, tym więcej pracy musi wykonać oprogramowanie bazy danych w celu ukończenia instrukcji SQL.



Jeżeli jest używana baza danych w wersji wcześniejszej niż Oracle Database 10g, wyniki całkowitego kosztu wykonania instrukcji mogą być puste. Dzieje się tak, ponieważ wcześniejsze wersje bazy danych nie zbierają automatycznie statystyk tabel. Do zbierania statystyk należy użyć polecenia ANALYZE. Informacje na ten temat znajdują się w podrozdziale „Zbieranie statystyk tabeli”.

Plany wykonywania dla złączeń tabel

Plany wykonywania zapytań zawierających złączenia tabel są bardziej złożone. Poniższy przykład generuje plan wykonania dla zapytania łączącego tabele products i product_types:

```
EXPLAIN PLAN SET STATEMENT_ID = 'PRODUCTS' FOR
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id;
```

Plan wykonywania tego zapytania został przedstawiony poniżej:

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: PRODUCTS
```

EXECUTION_PLAN

```
0 SELECT STATEMENT Koszt = 6
1 MERGE JOIN Koszt = 1
2 TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Koszt = 1
3 INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Koszt = 1
4 SORT JOIN Koszt = 2
5 TABLE ACCESS FULL PRODUCTS TABLE Koszt = 1
```

 Wyniki otrzymane po uruchomieniu tego zapytania mogą być inne w zależności od wersji używanej bazy danych oraz ustawień parametrów w pliku konfiguracyjnym *init.ora*.

Powyższy plan wykonania jest bardziej złożony i możemy w nim zauważyc hierarchiczne relacje między różnymi operacjami. Operacje są wykonywane w kolejności: 3, 2, 5, 4, 1 i 0. W tabeli 16.2 zostały opisane poszczególne operacje w kolejności ich wykonywania.

Tabela 16.2. Operacje w planie wykonania

ID Operacji	Opis
3	Pełne skanowanie indeksu product_types_pk (który jest indeksem unikatowym) w celu uzyskania adresów wierszy w tabeli product_types. Adresy mają postać wartości ROWID i są przesyłane do operacji 2
2	Dostęp do wierszy w tabeli product_types z użyciem listy wartości ROWID przesyłanych z operacji 3. Wiersze są przesyłane do operacji 1
5	Dostęp do wierszy w tabeli products. Wiersze są przesyłane do operacji 4
4	Sortowanie wierszy przesyłanych przez operację 5. Posortowane wiersze są przesyłane do operacji 1
1	Scalenie wierszy przesyłanych z operacji 2 i 5. Scalone wiersze są przesyłane do operacji 0
0	Zwrócenie wierszy z operacji 1 do użytkownika. Całkowity koszt zapytania wynosi 6 jednostek pracy

Zbieranie statystyk tabeli

Jeżeli używasz bazy danych w wersji wcześniejszej niż Oracle Database 10g, statystyki tabeli należy zbierać samodzielnie za pomocą polecenia ANALYZE. Domyślnie, jeżeli nie są dostępne żadne statystyki, używana jest optymalizacja oparta na regułach. Ten rodzaj optymalizacji zwykle nie jest tak dobry jak optymalizacja oparta na koszcie.

W poniższych przykładach użyto polecenia ANALYZE do zbierania statystyk tabel products i product_types:

```
ANALYZE TABLE products COMPUTE STATISTICS;
ANALYZE TABLE product_types COMPUTE STATISTICS;
```

Po zebraniu statystyk będzie wykorzystywana optymalizacja oparta na koszcie, a nie na regułach.

Porównywanie planów wykonania

Porównując całkowity koszt przedstawiony w planie wykonania różnych instrukcji SQL, możemy określić wartość optymalizacji instrukcji SQL. Z tego podrozdziału dowiesz się, jak porównać dwa plany wykonania i jakie korzyści płyną z zastosowania operatora EXISTS zamiast opcji DISTINCT. Poniższy przykład generuje plan wykonania zapytania wykorzystującego operator EXISTS:

```
EXPLAIN PLAN SET STATEMENT_ID = 'EXISTS_QUERY' FOR
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
```

```
FROM purchases inner
WHERE inner.product_id = outer.product_id);
```

Plan wykonania tego zapytania został przedstawiony poniżej. Można zauważyć, że koszt tego zapytania to 5 jednostek pracy:

```
@ c:\sql_book\sql\explain_plan.sql
Proszę podać wartość dla v_statement_id: EXISTS_QUERY
```

EXECUTION_PLAN

```
0 SELECT STATEMENT    Koszt = 5
1   MERGE JOIN SEMI   Koszt = 1
2     TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Koszt = 1
3       INDEX FULL SCAN PRODUCTS_PK INDEX (UNIQUE) Koszt = 1
4     SORT UNIQUE   Koszt = 2
5       INDEX FULL SCAN PURCHASES_PK INDEX (UNIQUE) Koszt = 1
```

Kolejny przykład generuje plan wykonania dla zapytania wykorzystującego DISTINCT:

```
EXPLAIN PLAN SET STATEMENT_ID = 'DISTINCT_QUERY' FOR
SELECT DISTINCT pr.product_id, pr.name
FROM products pr, purchases pu
WHERE pr.product_id = pu.product_id;
```

Plan wykonania tego zapytania został przedstawiony poniżej. Można zauważyć, że koszt tego zapytania to 6 jednostek pracy:

```
@ c:\sql_book\sql\explain_plan.sql
Proszę podać wartość dla v_statement_id: DISTINCT_QUERY
```

EXECUTION_PLAN

```
0 SELECT STATEMENT    Koszt = 6
1   HASH UNIQUE   Koszt = 1
2     MERGE JOIN   Koszt = 1
3       TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Koszt = 1
4         INDEX FULL SCAN PRODUCTS_PK INDEX (UNIQUE) Koszt = 1
5     SORT JOIN   Koszt = 2
6       INDEX FULL SCAN PURCHASES_PK INDEX (UNIQUE) Koszt = 1
```

To zapytanie jest bardziej kosztowne niż zapytanie wcześniejsze, w którym użyto EXISTS (koszt tego zapytania wynosił 5 jednostek). Te wyniki udowadniają, że stosowanie operatora EXISTS jest bardziej wydajne od DISTINCT.

Przesyłanie wskazówek do optymalizatora

Do optymalizatora można przesyłać wskazówki. Wskazówka jest dyrektywą optymalizatora, która wpływa na wybór planu wykonania przez optymalizator. Prawidłowa wskazówka może zwiększyć wydajność instrukcji SQL. Wydajność wskazówek można sprawdzić przez porównanie kosztów w planach wykonania instrukcji SQL ze wskazówką i bez niej.

W tym podrozdziale zajmiemy się przykładowym zapytaniem, w którym użyto jednej z najbardziej przydatnych wskazówek — FIRST_ROWS(n). Nakazuje ona optymalizatorowi wygenerowanie planu wykonania minimalizującego czas potrzebny do zwrócenia pierwszych n wierszy zapytania. Ta wskazówka może być przydatna, gdy nie chcemy czekać zbyt długo na uzyskanie jakichś wierszy z zapytania, ale i tak chcemy pobrać wszystkie wiersze.

Poniższy przykład generuje plan wykonania dla zapytania, w którym użyto wskazówki FIRST_ROWS(2); należy zauważać, że wskazówka została umieszczona między napisami /*+ i */:

```
EXPLAIN PLAN SET STATEMENT_ID = 'HINT' FOR
SELECT /*+ FIRST_ROWS(2) */ p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id;
```



Składnia wskazówek musi być zgodna z przedstawioną powyżej. W przeciwnym razie wskazówka zostanie pominięta. Składnia wskazówek ma postać: /*+ jedna spacja, treść wskazówki, jedna spacja i */.

Plan wykonania tego zapytania został przedstawiony poniżej. Należy zauważyć, że koszt wynosi 4 jednostki pracy:

```
@ c:\sql_book\sql\explain_plan.sql
Proszę podać wartość dla v_statement_id: HINT

EXECUTION_PLAN
-----
0 SELECT STATEMENT Koszt = 4
1   NESTED LOOPS
2     NESTED LOOPS Koszt = 1
3       TABLE ACCESS FULL PRODUCTS TABLE Koszt = 1
4         INDEX UNIQUE SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Koszt = 2
5       TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Koszt = 2
```

Kolejny przykład generuje plan wykonania tego samego zapytania bez wskazówek:

```
EXPLAIN PLAN SET STATEMENT_ID = 'NO_HINT' FOR
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id;
```

Plan wykonania tego zapytania został przedstawiony poniżej. Należy zauważyć, że koszt wynosi 6 jednostek pracy (czyli jest wyższy niż zapytania ze wskazówką):

```
@ c:\sql_book\sql\explain_plan.sql
Proszę podać wartość dla v_statement_id: NO_HINT
```

```
EXECUTION_PLAN
-----
0 SELECT STATEMENT Koszt = 6
1   MERGE JOIN Koszt = 1
2     TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Koszt = 1
3       INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Koszt = 1
4     SORT JOIN Koszt = 2
5       TABLE ACCESS FULL PRODUCTS TABLE Koszt = 1
```

Te wyniki pokazują, że dołączenie wskazówki zmniejsza koszt uruchomienia zapytania o 2 jednostki pracy.

Istnieje wiele różnych wskazówek, a ten rozdział stanowi jedynie krótkie wprowadzenie do tego tematu. Wskazówek używa się sporadycznie, ponieważ Oracle Corporation dostarcza wielu dobrych narzędzi do optymalizacji wydajności, które są łatwiejsze w użyciu niż wskazówki i zazwyczaj są bardziej efektywne. W kolejnych podrozdziałach znajduje się wprowadzenie do tych narzędzi.

Dodatkowe narzędzia optymalizujące

W tym ostatnim podrozdziale zostaną krótko opisane inne narzędzia optymalizacyjne. Ich szczegółowy opis wykracza poza zakres tej książki. Dokładne omówienie tych narzędzi można znaleźć w podręczniku *Oracle Database Performance Tuning Guide* opublikowanym przez Oracle Corporation.

Oracle Enterprise Manager

Oracle Enterprise Manager zawiera wiele komponentów do analizy wydajności, które przechwytyują dane o systemie operacyjnym, warstwie pośredniczącej oraz dane o wydajności aplikacji i bazy danych. Oracle Enterprise Manager analizuje informacje o wydajności i prezentuje wyniki w postaci graficznej. Administrator bazy danych może również skonfigurować Oracle Enterprise Manager tak, aby alarmował o problemach z wydajnością za pomocą poczty elektronicznej lub wiadomości tekstowych. Oracle Enterprise Manager zawiera również wskazówki programowe pomagające rozwiązać problemy z wydajnością.

Automatic Database Diagnostic Monitor

Automatic Database Diagnostic Monitor jest modułem diagnostycznym wbudowanym do oprogramowania bazy danych Oracle. ADDM umożliwia administratorowi bazy danych określenie problemów związanych z wydajnością bazy danych przez analizę wydajności systemu w długim okresie. Administrator bazy danych może przeglądać informacje generowane przez ADDM w Oracle Enterprise Manager. Jeżeli ADDM wykryje problemy z wydajnością, zasugeruje możliwe rozwiązania, w tym na przykład:

- zmiany sprzętowe — na przykład dodanie procesorów do serwera bazy danych,
- konfigurację bazy danych — na przykład zmianę ustawień parametrów inicjalizujących,
- zmiany w aplikacji — na przykład użycie opcji buforowania dla sekwencji lub użycie zmiennych dowiązanych,
- użycie innych programów doradzających — na przykład uruchomienie SQL Tuning Advisor i SQL Access Advisor dla instrukcji SQL, których wykonanie zużywa najwięcej zasobów bazy danych.

SQL Tuning Advisor

SQL Tuning Advisor umożliwia programistie lub administratorowi bazy danych optymalizację instrukcji SQL za pomocą następujących elementów:

- tekstu instrukcji SQL,
- identyfikatora SQL instrukcji (uzyskanego z perspektywy V\$SQL_PLAN, która jest jedną z perspektyw dostępnych dla administratora bazy danych),
- zakresu identyfikatorów migawek,
- nazwy SQL Tuning Set.

SQL Tuning Set jest zestawem instrukcji SQL przypisanym planem wykonania i statystykami wykonania. Zestawy SQL Tuning Set są analizowane w celu wygenerowania profilów SQL ułatwiających optymalizatorowi wybranie optymalnego planu wykonania.

SQL Access Advisor

SQL Access Advisor dostarcza administratorowi lub programistie wskazówki dotyczące perspektyw zmaterializowanych, indeksów oraz dzienników zmaterializowanych perspektyw. SQL Access Advisor bada użycie przestrzeni oraz wydajność zapytań i zaleca najbardziej wydajną (pod względem kosztów) konfigurację dla nowych i istniejących zmaterializowanych perspektyw i indeksów.

SQL Performance Analyzer

SQL Performance Analyzer umożliwia sprawdzanie wpływu zmian wprowadzanych w systemie na wydajność serwera SQL poprzez określanie instrukcji SQL, których wydajność spadła, wzrosła lub pozostała na tym samym poziomie.

Database Replay

Database Replay umożliwia zapisanie obciążenia bazy danych z systemu produkcyjnego. Następnie można odtworzyć identyczne obciążenie w systemie testowym z takimi samymi zależnościami czasowymi i współbieżnością jak w systemie produkcyjnym — dla porównania. Można taki zapis odtworzyć na tej samej lub nowszej wersji bazy danych Oracle.

Real-Time SQL Monitoring

Real-Time SQL Monitoring umożliwia monitorowanie wydajności instrukcji SQL podczas ich wykonywania. Domyślnie monitorowanie uruchamiane jest automatycznie, gdy instrukcje SQL wykonywane są

równolegle lub gdy ich wykonanie zajęło co najmniej 5 sekund czasu procesora lub czasu operacji wejścia-wyjścia podczas jednego wykonania.

SQL Plan Management

SQL Plan Management zapisuje i ocenia plany wykonania. Baza danych tworzy zestaw podstawowych planów wykonania. Jeśli wykonanie tej samej instrukcji SQL jest powtarzane i optymalizator tworzy nowy plan różny od podstawowego, baza danych porównuje nowy plan z podstawowym i wykorzystuje najefektywniejszy.

Podsumowanie

Z tego rozdziału dowiedziałeś się, że:

- optymalizacja jest procesem powodującym szybsze wykonywanie instrukcji SQL,
- optymalizator jest podsystemem oprogramowania bazy danych Oracle, który generuje plan wykonania będący zestawem operacji koniecznych do wykonania określonej instrukcji SQL,
- do optymalizatora można przesyłać wskazówki, a przez to wpływać na generowany plan wykonania dla instrukcji SQL,
- istnieje wiele dodatkowych narzędzi umożliwiających administratorowi bazy danych optymalizację bazy danych.

W następnym rozdziale zajmiemy się XML.

ROZDZIAŁ

17

XML i baza danych Oracle

Ten rozdział zawiera krótkie wprowadzenie do XML oraz pokazuje, jak:

- generować XML z danych relacyjnych,
- zapisać XML w bazie danych.

Wprowadzenie do XML

Extensible Markup Language (XML) jest językiem znaczników o ogólnym zastosowaniu. XML umożliwia wymianę ustrukturyzowanych danych w internecie i może być używany do kodowania danych i innych dokumentów.

Zalety XML to między innymi to, że:

- może być odczytany zarówno przez ludzi, jak i komputery, a oprócz tego jest składowany jako zwykły tekst,
- jest niezależny od platformy,
- obsługuje Unicode, co oznacza, że może zawierać informacje napisane w dowolnym ludzkim języku,
- wykorzystuje samoopisujący się format, zawierający strukturę dokumentu, nazwy elementów i wartości elementów.

Dzięki tym zaletom XML jest często stosowany przy składowaniu i przetwarzaniu dokumentów. Jest wykorzystywany przez wiele organizacji do wymiany danych między systemami komputerowymi. Na przykład wielu dostawców umożliwia klientom przesyłanie zamówień w postaci plików XML przez internet.

Poniższa lista pokazuje skróconą historię obsługi XML w bazie danych Oracle:

- W Oracle Database 9i wprowadzono możliwość składowania XML w bazie danych oraz wiele rozbudowanych funkcji do manipulowania i przetwarzania XML.
- W Oracle Database Release 2 dodano funkcje generujące XML
- W Oracle Database 11g dodano możliwości przetwarzania danych binarnych XML w Java i C (binarny XML zapewnia bardziej wydajne składowanie i manipulowanie XML w bazie danych).
- Oracle Database 12c jeszcze bardziej rozszerza funkcjonalności związane z XML o dodatkowe pakiety PL/SQL wspierające XML, XQuery API dla języka Java (XQJ) i inne. W tej wersji Oracle Database funkcjonalność Oracle XML DB nie może być odinstalowana.

W tym rozdziale skupimy się na przydatnym zestawie możliwości obsługi XML w bazie danych Oracle. Dużo informacji na temat XML znajduje się w poniższych witrynach:

- <http://www.w3.org/XML>,
- <http://pl.wikipedia.org/wiki/XML>.

Generowanie XML z danych relacyjnych

Baza danych Oracle posiada kilka funkcji umożliwiających generowanie XML. Z tego podrozdziału dowiesz się, jak można za ich pomocą wygenerować XML z danych relacyjnych.

XMLEMENT()

Funkcja XMLEMENT() służy do generowania elementów XML z danych relacyjnych. Należy przesłać nazwę elementu oraz kolumnę, którą chcemy pobrać do XMLEMENT(). Funkcja zwraca elementy jako obiekty XMLType. Typ XMLType jest wbudowanym typem bazy danych, który służy do reprezentacji danych XML.

Poniższy przykład nawiązuje połączenie jako użytkownik `store` i pobiera wartości kolumny `customer_id` jako obiekty XMLType:

```
CONNECT store/store_password
SELECT XMLEMENT("customer_id", customer_id)
AS xml_customers
FROM customers;
```

XML_CUSTOMERS

```
-----
<customer_id>1</customer_id>
<customer_id>2</customer_id>
<customer_id>3</customer_id>
<customer_id>4</customer_id>
<customer_id>5</customer_id>
```

W wynikach XMLEMENT("customer_id", `customer_id`) zwraca wartości `customer_id` w znaczniku `customer_id`. Można użyć dowolnej nazwy znacznika, co obrazuje poniższy przykład, w którym zastosowano znacznik "cust_id":

```
SELECT XMLEMENT("cust_id", customer_id)
AS xml_customers
FROM customers;
```

XML_CUSTOMERS

```
-----
<cust_id>1</cust_id>
<cust_id>2</cust_id>
<cust_id>3</cust_id>
<cust_id>4</cust_id>
<cust_id>5</cust_id>
```

Kolejny przykład pobiera wartości `first_name` i `dob` klienta nr 2:

```
SELECT XMLEMENT("first_name", first_name) || XMLEMENT("dob", dob)
AS xml_customer
FROM customers
WHERE customer_id = 2;
```

XML_CUSTOMER

```
-----
<first_name>Lidia</first_name><dob>1968-02-05</dob>
```

W poniższym przykładzie użyto funkcji `TO_CHAR()` w celu zmienienia formatu danych wartości `dob`:

```
SELECT XMLEMENT("dob", TO_CHAR(dob, 'MM/DD/YYYY'))
AS xml_dob
FROM customers
WHERE customer_id = 2;
```

XML_DOB

```
-----
<dob>02/05/1968</dob>
```

W kolejnym przykładzie osadzono dwa wywołania XMLEMENT() w zewnętrznym wywołaniu funkcji XMLEMENT(). Należy zauważyć, że zwrócone elementy `customer_id` i `name` znajdują się wewnątrz zewnętrznego elementu `customer`:

```
SELECT XMLEMENT(
    "customer",
    XMLEMENT("customer_id", customer_id),
    XMLEMENT("name", first_name || ' ' || last_name)
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);
```

`XML_CUSTOMERS`

```
-----  
<customer>  
  <customer_id>1</customer_id>  
  <name>Jan Nikiel</name>  
</customer>  
  
<customer>  
  <customer_id>2</customer_id>  
  <name>Lidia Stal</name>  
</customer>
```



W XML zwróconym przez powyższe zapytanie dodano podziały wierszy i odstępów, aby kod był czytelniejszy. Dotyczy to również innych przykładów prezentowanych w tym rozdziale. W wyniku zapytania zazwyczaj otrzymuje się XML w jednej linii bez żadnych znaków nowej linii.

Można pobierać zarówno zwykłe dane relacyjne, jak i XML, co obrazuje poniższy przykład pobierający kolumnę `customer_id` jako zwykły wynik relacyjny oraz kolumny `first_name` i `last_name` skonkatenowane jako elementy XML:

```
SELECT customer_id,
       XMLEMENT("customer", first_name || ' ' || last_name) AS xml_customer
  FROM customers;
```

`CUSTOMER_ID XML_CUSTOMER`

```
-----  
  1 <customer>Jan Nikiel</customer>  
  2 <customer>Lidia Stal</customer>  
  3 <customer>Stefan Brąz</customer>  
  4 <customer>Grażyna Cynk</customer>  
  5 <customer>Jadwiga Mosiądz</customer>
```

Można również generować XML dla obiektów bazy danych, co obrazuje poniższy przykład, nawiązujący połączenie jako `object_user` i pobierający kolumny `id` i `address` klienta nr 1 z tabeli `object_customers` (w kolumnie `address` jest składowany obiekt o typie `t_address`):

```
CONNECT object_user/object_password
SELECT XMLEMENT("id", id) || XMLEMENT("address", address)
  AS xml_object_customer
  FROM object_customers
 WHERE id = 1;
```

`XML_OBJECT_CUSTOMER`

```
-----  
<id>1</id>  
<address>  
  <T_ADDRESS>  
    <STREET>Kościuszki 23</STREET>  
    <CITY>Siemiatycze</CITY>  
    <STATE>MAL</STATE>
```

```
<ZIP>12345</ZIP>
</T_ADDRESS>
</address>
```

Można również generować XML na podstawie kolekcji, co obrazuje poniższy przykład, nawiązujący połączenie jako użytkownik `collection_user` i pobierający wartości kolumn `id` i `addresses` klienta nr 1, składowane w tabeli `customers_with_nested_table` (w kolumnie `addresses` jest składowany obiekt typu `t_nested_table`, będący tabelą zagnieżdżoną z obiektami `t_address`):

```
CONNECT collection_user/collection_password
SELECT XMLEMENT("id", id) || XMLEMENT("addresses", addresses)
AS xml_customer
FROM customers_with_nested_table
WHERE id = 1;
```

```
XML_CUSTOMER
-----
<id>1</id>
<addresses>
  <T_NESTED_TABLE_ADDRESS>
    <T_ADDRESS>
      <STREET>Stanowa 2</STREET><CITY>Fasolowo</CITY>
      <STATE>MAZ</STATE><ZIP>12345</ZIP>
    </T_ADDRESS>
    <T_ADDRESS>
      <STREET>Górska 4</STREET>
      <CITY>Rzeszotary</CITY>
      <STATE>GDA</STATE>
      <ZIP>54321</ZIP>
    </T_ADDRESS>
  </T_NESTED_TABLE_ADDRESS>
</addresses>
```

XMLATTRIBUTES()

Funkcja `XMLATTRIBUTES()` jest używana w połączeniu z funkcją `XMLEMENT()` w celu określenia atrybutów elementów XML pobieranych przez `XMLEMENT()`. Poniższy przykład nawiązuje połączenie jako użytkownik `store` i wykorzystuje funkcję `XMLATTRIBUTES()` do ustawienia nazw atrybutów dla elementów `customer_id`, `first_name`, `last_name` i `dob`:

```
CONNECT store/store_password
SELECT XMLEMENT(
  "customer",
  XMLATTRIBUTES(
    customer_id AS "id",
    first_name || ' ' || last_name AS "name",
    TO_CHAR(dob, 'DD/MM/YYYY') AS "dob"
  )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);
```

```
XML_CUSTOMERS
-----
<customer id="1" name="Jan Nikiel" dob="01/01/1965"></customer>
<customer id="2" name="Lidia Stal" dob="05/02/1968"></customer>
```

Należy zauważać, że atrybuty `id`, `name` i `dob` zostały zwrócone wewnętrz znacznika `customer`.

XMLFOREST()

Funkcja `XMLFOREST()` służy do generowania „lasu” elementów XML. Konkatenuje elementy XML, więc nie jest konieczne używanie operatora konkatenacji `||` z wieloma wywołaniami funkcji `XMLEMENT()`.

W poniższym przykładzie użyto funkcji XMLFOREST() do pobrania wartości customer_id, phone i dob klientów nr 1 i 2:

```
SELECT XMLEMENT(
  "customer",
  XMLFOREST(
    customer_id AS "id",
    phone AS "phone",
    TO_CHAR(dob, 'DD/MM/YYYY') AS "dob"
  )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer>
  <id>1</id>
  <phone>800-555-1211</phone>
  <dob>01/01/1965</dob>
</customer>

<customer>
  <id>2</id>
  <phone>800-555-1212</phone>
  <dob>05/02/1968</dob>
</customer>
```

Poniższe polecenie przypisuje parametrowi LONG SQL*Plus wartość 500, dzięki czemu będzie widoczny cały kod XML zwracany przez kolejne zapytania (parametr LONG określa maksymalną długość danych tekstowych wyświetlanych przez SQL*Plus):

```
SET LONG 500
```

Poniższe zapytanie umieszcza nazwę klienta w znaczniku elementu customer, korzystając z funkcji XMLATTRIBUTES():

```
SELECT XMLEMENT(
  "customer",
  XMLATTRIBUTES(first_name || ' ' || last_name AS "name"),
  XMLFOREST(phone AS "phone", TO_CHAR(dob, 'DD/MM/YYYY') AS "dob")
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer name="Jan Nikiel">
  <phone>800-555-1211</phone>
  <dob>01/01/1965</dob>
</customer>

<customer name="Lidia Stal">
  <phone>800-555-1212</phone>
  <dob>05/02/1968</dob>
</customer>
```

XMLAGG()

Funkcja XMLAGG() generuje las elementów XML z kolekcji elementów XML. Jest zwykle używana do grupowania XML w jednorodną listę elementów pod jednym rodzicem lub do pobierania danych z kolekcji. W celu pogrupowania zwróconego zestawu wierszy można użyć w zapytaniu klauzuli GROUP BY, do posortowania wierszy można natomiast użyć klauzuli ORDER BY funkcji XMLAGG().

Domyślnie klauzula ORDER BY sortuje wyniki rosnąco, ale dodanie opcji DESC po liście kolumn do sortowania posortuje wiersze w porządku malejącym. Można również dodać opcję ASC, aby jawnie posortować wiersze rosnąco. Opcja NULLS LAST umieszcza wszelkie wartości NULL na końcu wyników.

Poniższy przykład pobiera wartości first_name i last_name klienta i zwraca je w postaci listy o nazwie customer_list. Należy zwrócić uwagę, że w funkcji XMLAGG() użyto klauzuli ORDER BY w celu posortowania wyników według kolumny name. Dodano również opcję ASC, aby jawnie wskazać rosnący porządek sortowania:

```
SELECT XMLEMENT(
  "customer_list",
  XMLAGG(
    XMLEMENT("customer", first_name || ' ' || last_name)
    ORDER BY first_name ASC
  )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer_list>
  <customer>Jan Nikiel</customer>
  <customer>Lidia Stal</customer>
</customer_list>
```

Kolejny przykład pobiera wartości product_type_id oraz średnią cenę (price) każdej grupy produktów. Należy zwrócić uwagę, że produkty zostały pogrupowane według product_type_id za pomocą klauzuli GROUP BY zapytania. Ponadto w klauzuli ORDER BY funkcji XMLAGG() użyto opcji NULLS LAST, aby umieścić na końcu wyników wiersz, w którym product_type_id ma wartość NULL:

```
SELECT XMLEMENT(
  "product_list",
  XMLAGG(
    XMLEMENT(
      "product_type_and_avg", product_type_id || ' ' || AVG(price)
    )
    ORDER BY product_type_id NULLS LAST
  )
)
AS xml_products
FROM products
GROUP BY product_type_id;

XML_PRODUCTS
-----
<product_list>
  <product_type_and_avg>1 24,975</product_type_and_avg>
  <product_type_and_avg>2 26,22</product_type_and_avg>
  <product_type_and_avg>3 13,24</product_type_and_avg>
  <product_type_and_avg>4 13,99</product_type_and_avg>
  <product_type_and_avg> null</product_type_and_avg>
</product_list>
```



Wartości NULL mogą być również umieszczane jako pierwsze. W tym celu należy użyć opcji NULLS FIRST w klauzuli ORDER BY.

Kolejny przykład pobiera wartości product_type_id i name produktów, których product_type_id ma wartość 1 lub 2. Produkty są grupowane według product_type_id:

```
SELECT XMLEMENT(
  "products_in_group",
  XMLATTRIBUTES(product_type_id AS "prd_type_id"),
```

```

XMLAGG(
    XMLELEMENT("name", name)
)
)
AS xml_products
FROM products
WHERE product_type_id IN (1, 2)
GROUP BY product_type_id;

XML_PRODUCTS
-----
<products_in_group prd_type_id="1">
    <name>Nauka współczesna</name>
    <name>Chemia</name>
</products_in_group>

<products_in_group prd_type_id="2">
    <name>Supernowa</name>
    <name>2412: Powrót</name>
    <name>Z Files</name>
    <name>Wojny czołgów</name>
</products_in_group>

```

Kolejny przykład nawiązuje połączenie jako `collection_user` i pobiera adresy klienta nr 1 z tabeli `customers_with_nested_table`:

```

CONNECT collection_user/collection_password
SELECT XMLELEMENT("customer",
    XMLAGG(
        XMLELEMENT("addresses", addresses)
    )
)
AS xml_customer
FROM customers_with_nested_table
WHERE id = 1;

XML_CUSTOMER
-----
<customer>
    <addresses>
        <T_NESTED_TABLE_ADDRESS>
            <T_ADDRESS>
                <STREET>Stanowa 2</STREET>
                <CITY>Fasolowo</CITY>
                <STATE>MAZ</STATE>
                <ZIP>12345</ZIP>
            </T_ADDRESS>
            <T_ADDRESS>
                <STREET>Górska 4</STREET>
                <CITY>Rzeszotary</CITY>
                <STATE>GDA</STATE>
                <ZIP>54321</ZIP>
            </T_ADDRESS>
        </T_NESTED_TABLE_ADDRESS>
    </addresses>
</customer>

```

XMLCOLATTVAL()

Funkcja `XMLCOLATTVAL()` służy do utworzenia fragmentu XML i późniejszego rozszerzenia powstałego XML. Każdy fragment XML posiada tę samą nazwę kolumny w nazwie atrybutu. Nazwę atrybutu można zmienić za pomocą klauzuli AS.

Poniższy przykład nawiązuje połączenie jako użytkownik `store` i pobiera wartości kolumn `customer_id`, `dob` i `phone` dla klientów nr 1 i 2:

```
CONNECT store/store_password
SELECT XMLELEMENT(
    "customer",
    XMLCOLATTVAL(
        customer_id AS "id",
        dob AS "dob",
        phone AS "phone"
    )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer>
  <column name = "id">1</column>
  <column name = "dob">1965-01-01</column>
  <column name = "phone">800-555-1211</column>
</customer>

<customer>
  <column name = "id">2</column>
  <column name = "dob">1968-02-05</column>
  <column name = "phone">800-555-1212</column>
</customer>
```

XMLCONCAT()

Funkcja `XMLCONCAT()` konkatenuje serię elementów każdego wiersza. Poniższy przykład konkatenuje elementy XML dla wartości `first_name`, `last_name` i `phone` klientów nr 1 i 2:

```
SELECT XMLCONCAT(
    XMLEMENT("first name", first_name),
    XMLEMENT("last name", last_name),
    XMLEMENT("phone", phone)
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<first name>Jan</first name>
<last name>Nikiel</last name>
<phone>800-555-1211</phone>

<first name>Lidia</first name>
<last name>Stal</last name>
<phone>800-555-1212</phone>
```

XMLPARSE()

Funkcja `XMLPARSE()` służy do parsowania i generowania XML z obliczonego wyniku wyrażenia. Wyrażenie musi zwracać napis. Jeżeli zwróci `NULL`, funkcja `XMLPARSE()` również zwróci `NULL`. Przed wyrażeniem należy określić jeden z poniższych elementów:

- **CONTENT** oznacza, że wynikiem wyrażenia musi być poprawna wartość XML,
- **DOCUMENT** oznacza, że wynikiem wyrażenia musi być dokument XML o jednym korzeniu.

Po wyrażeniu można również umieścić opcję WELLFORMED, co oznacza, że mamy pewność, iż nasze wyrażenie daje w wyniku poprawny składniowo dokument XML. To również oznacza, że baza danych nie przeprowadzi testów poprawności tego wyrażenia.

Poniższy przykład parsuje wyrażenie zawierające informacje o kliencie:

```
SELECT XMLPARSE(
  CONTENT
  '<customer><customer_id>1</customer_id><name>Jan Brązowy
   </name></customer>'
  WELLFORMED
)
AS xml_customer
FROM dual;
```

XML_CUSTOMER

```
-----
<customer>
  <customer_id>1</customer_id>
  <name>Jan Brązowy </name>
</customer>
```



Więcej informacji na temat poprawnych składniowo dokumentów XML i wartości można znaleźć pod adresem <http://www.w3.org/TR/REC-xml>.

XMLPI()

Funkcja XMLPI() służy do generowania instrukcji przetwarzania XML. Są one zwykle używane do dostarczenia aplikacji informacji powiązanych z danymi XML. Wówczas aplikacja może użyć instrukcji przetwarzania do określenia, jak przetworzyć te dane.

Poniższy przykład generuje instrukcję przetwarzania dla statusu zamówienia:

```
SELECT XMLPI(
  NAME "order_status",
  'ZŁOŻONE, OCZEKUJĄCE, WYSŁANE'
)
AS xml_order_status_pi
FROM dual;
```

XML_ORDER_STATUS_PI

```
-----
<?order_status ZŁOŻONE, OCZEKUJĄCE, WYSŁANE?>
```

Kolejny przykład generuje instrukcję przetwarzania wyświetlającą dokument XML z użyciem kaskadowego arkusza stylów o nazwie *example.css*:

```
SELECT XMLPI(
  NAME "xml-stylesheet",
  'type="text/css" href="example.css"'
)
AS xml_stylesheet_pi
FROM dual;
```

XML_STYLESHEET_PI

```
-----
<?xml-stylesheet type="text/css" href="example.css"?>
```

XMLCOMMENT()

Funkcja XMLCOMMENT() generuje komentarz XML, czyli napis umieszczony między znacznikami <!-- i -->. Na przykład:

```

SELECT XMLCOMMENT(
  'Przykładowy komentarz XML'
)
AS xml_comment
FROM dual;

XML_COMMENT
-----
<!--Przykładowy komentarz XML-->

```

XMLSEQUENCE()

Funkcja XMLSEQUENCE() generuje obiekt XMLSequenceType będący tabelą o zmiennej długości, zawierającą obiekty XMLType. Ponieważ funkcja XMLSEQUENCE() zwraca VARRAY, możemy jej użyć w klauzuli FROM zapytania. Na przykład:

```

SELECT VALUE(list_of_values).GETSTRINGVAL() order_values
FROM TABLE(
  XMLSEQUENCE(
    EXTRACT(
      XMLType('<A><B>ZŁOŻONE</B><B>OCZEKUJĄCE</B><B>WYSŁANE</B></A>'),
      '/A/B'
    )
  )
) list_of_values;

ORDER_VALUES
-----
<B>ZŁOŻONE</B>
<B>OCZEKUJĄCE</B>
<B>WYSŁANE</B>

```

Omówmy kolejne fragmenty tego przykładu. Wywołanie funkcji XMLType() ma postać:

```
XMLType ('<A><B>ZŁOŻONE</B><B>OCZEKUJĄCE</B><B>WYSŁANE</B></A>')
```

Tworzy ono obiekt XMLType zawierający kod XML:

```
<B>ZŁOŻONE</B><B>OCZEKUJĄCE</B><B>WYSŁANE</B></A>
```

Wywołanie funkcji EXTRACT() ma postać:

```

EXTRACT(
  XMLType('<A><B>ZŁOŻONE</B><B>OCZEKUJĄCE</B><B>WYSŁANE</B></A>'),
  '/A/B'
)

```

Funkcja EXTRACT() wydobywa dane XML z obiektu XMLType zwróconego przez wywołanie funkcji XMLType(). Drugim parametrem funkcji EXTRACT() jest napis XPath. XPath jest językiem umożliwiającym uzyskanie dostępu do określonych elementów w danych XML. Na przykład w naszym przykładowym wywoaniu EXTRACT() '/A/B' zwraca wszystkie elementy B będące elementami potomnymi elementów A, dlatego też funkcja EXTRACT() zwraca następujące wartości:

```
<B>ZŁOŻONE</B>
<B>OCZEKUJĄCE</B>
<B>WYSŁANE</B>
```

Wywołanie funkcji XMLSEQUENCE() w naszym przykładzie po prostu zwraca tablicę o zmiennej długości, zawierającą elementy zwrócone przez funkcję EXTRACT(). Funkcja TABLE() konwertuje VARRAY na tablicę wierszy i nadaje jej alias list_of_values. Instrukcja SELECT pobiera wartości napisów z wierszy tabeli, korzystając z metody GETSTRINGVAL().

W dalszej części rozdziału zostaną przedstawione inne przykłady użycia funkcji EXTRACT() i języka XPath.

XMLSERIALIZE()

Funkcja XMLSERIALIZE() generuje napis lub LOB (duży obiekt) stanowiący reprezentację danych XML na podstawie obliczonego wyniku wyrażenia. Przed wyrażeniem należy określić jeden z poniższych elementów:

- CONTENT oznacza, że wynikiem wyrażenia musi być poprawna wartość XML,
- DOCUMENT oznacza, że wynikiem wyrażenia musi być dokument XML o jednym korzeniu.

W poniższym przykładzie użyto funkcji XMLSERIALIZE z opcją CONTENT do wygenerowania wartości XML:

```
SELECT XMLSERIALIZE(
  CONTENT XMLType('<order_status>WYSŁANE</order_status>')
)
AS xml_order_status
FROM DUAL;
```

XML_ORDER_STATUS

<order_status>WYSŁANE</order_status>

W kolejnym przykładzie użyto funkcji XMLSERIALIZE() z opcją DOCUMENT w celu wygenerowania dokumentu XML zwracanego w obiekcie CLOB (dużym obiekcie znakowym):

```
SELECT XMLSERIALIZE(
  DOCUMENT XMLType('<description>Opis produktu</description>')
  AS CLOB
)
AS xml_product_description
FROM DUAL;
```

XML_PRODUCT_DESCRIPTION

<description>Opis produktu</description>

Przykład zapisywania danych XML do pliku w PL/SQL

W tym podrozdziale zostanie przedstawiony kompletny przykład PL/SQL zapisujący nazwy klientów do pliku XML. Musimy zacząć od nawiązania połączenia jako użytkownika z podwyższonymi uprawnieniami (na przykład system) i przydzielenia uprawnienia CREATE ANY DIRECTORY użytkownikowi store:

```
CONNECT system/oracle;
GRANT CREATE ANY DIRECTORY TO store;
```

Następnie łączymy się jako użytkownik store i tworzymy obiekt katalogu:

```
CONNECT store/store_password;
CREATE DIRECTORY TEMP_FILES_DIR AS 'C:\temp_files';
```

W powyższym przykładzie wpisana jest ścieżka z systemu Windows C:\temp_files. Należy również utworzyć katalog temp_files na partycji C:. Jeżeli jest używany system Unix lub Linux, należy utworzyć katalog na jednej z własnych partycji, a następnie użyć polecenia CREATE DIRECTORY z odpowiednią ścieżką dostępu. Należy się również upewnić, że konto użytkownika, z którego instalowano oprogramowanie bazy danych, ma uprawnienia zapisu do tego katalogu.

Następnie należy uruchomić skrypt *xml_examples.sql* znajdujący się w katalogu SQL:

```
@ C:\sql_book\SQL\xml_examples.sql
```

 **Ostrzeżenie** Należy uruchomić tylko skrypt *xml_examples.sql*. W katalogu SQL znajduje się jeszcze skrypt *xml_schema.sql*. Nie należy go jeszcze uruchamiać.

Skrypt *xml_examples.sql* tworzy dwie procedury. W tym podrozdziale zostanie opisana procedura *write_xml_data_to_file()*, która pobiera nazwy klientów i zapisuje je do pliku XML. Jest ona definiowana w następujący sposób:

```

CREATE PROCEDURE write_xml_data_to_file(
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_file UTL_FILE.FILE_TYPE;
    v_amount INTEGER := 32767;
    v_xml_data XMLType;
    v_char_buffer VARCHAR2(32767);
BEGIN
    -- otwiera plik do zapisu tekstu (maks. v_amount
    -- znaków w operacji)
    v_file := UTL_FILE.FOPEN(p_directory, p_file_name, 'w', v_amount);

    -- zapisuje początkowy wiersz do v_file
    UTL_FILE.PUT_LINE(v_file, '<?xml version="1.0"?>');

    -- pobiera inf o klientach i zapisuje je w v_xml_data
    SELECT
        EXTRACT(
            XMLELEMENT(
                "customer_list",
                XMLELEMENT("customer", first_name || ' ' || last_name)
                ORDER BY last_name
            )
        ),
        '/customer_list'
    )
    AS xml_customers
    INTO v_xml_data
    FROM customers;

    -- pobiera wartość napisu z v_xml_data i zapisuje ją w v_char_buffer
    v_char_buffer := v_xml_data.GETSTRINGVAL();

    -- kopiuje znaki z v_char_buffer do pliku
    UTL_FILE.PUT(v_file, v_char_buffer);

    -- zapisuje pozostałe dane w pliku
    UTL_FILE.FFLUSH(v_file);

    -- zamkna plik
    UTL_FILE.FCLOSE(v_file);
END write_xml_data_to_file;
/

```

Poniższa instrukcja wywołuje procedurę `write_xml_data_to_file()`:

```
CALL write_xml_data_to_file('TEMP_FILES_DIR', 'customers.xml');
```

Po uruchomieniu tej instrukcji w katalogu `C:\temp_files` (lub innym, który został określony we wcześniejszej instrukcji `CREATE DIRECTORY`) zostanie utworzony plik o nazwie `customers.xml`. Zawartość tego pliku jest następująca:

```
<?xml version="1.0"?>
<customer_list><customer>Stefan Brąz</customer><customer>Grażyna Cynk</customer><customer>Jadwiga
Mosiądz</customer><customer>Jan Nikiel</customer><customer>Lidia Stal</customer></customer_list>
```

Procedurę `write_xml_data_to_file()` można zmodyfikować tak, aby pobierała dowolne dane relacyjne i zapisywała je do pliku XML.

XMLQUERY()

Funkcja `XMLQUERY()` służy do konstruowania XML lub odpytywania XML. Do funkcji `XMLQUERY()` należy przesłać wyrażenie XQuery. XQuery jest językiem umożliwiającym konstruowanie i odpytywanie XML. Funkcja `XMLQUERY()` zwraca wynik przetworzenia wyrażenia XQuery.

Poniższy przykład obrazuje użycie funkcji XMLQUERY():

```
SELECT XMLQUERY(
  '(1, 2 + 5, "d", 155 to 161, <A>tekst</A>)'
  RETURNING CONTENT
)
AS xml_output
FROM DUAL;
```

XML_OUTPUT

```
-----  
1 7 d 155 156 157 158 159 160 161<A>tekst</A>
```

Oto kilka uwag dotyczących tego przykładu:

- Napis przesyłany do funkcji XMLQUERY() jest wyrażeniem XQuery, co oznacza, że wyrażenie XQuery ma postać `(1, 2 + 5, "d", 155 to 161, <A>tekst)`. Wartość 1 jest literałem liczby całkowitej, `2 + 5` jest wyrażeniem arytmetycznym, `d` jest napisem literałowym, `155 to 166` jest sekwencją liczb całkowitych, a `<A>tekst` jest elementem XML.
- Wszystkie elementy XQuery są przetwarzane kolejno. Na przykład przetwarzane jest `2 + 5` i zwarcane jest 7. Po przetworzeniu `155 to 161` zostaje zwrócone `155 156 157 158 159 160 161`.
- RETURNING CONTENT oznacza, że jest zwracany fragment XML. Stanowi on pojedynczy element XML z dowolną liczbą potomków, które również mogą być kodem XML dowolnego typu, w tym elementami tekstowymi. Fragment XML jest zgodny z rozszerzonym modelem danych Infoset (jest to specyfikacja opisująca abstrakcyjny model danych dokumentu XML; więcej informacji na ten temat można znaleźć pod adresem <http://www.w3.org/TR/xml-infoset>).

Przejdźmy do bardziej złożonego przykładu. Poniższa instrukcja (znajdująca się w skrypcie `xml_exam-ples.sql`) tworzy procedurę o nazwie `create_xml_resources()`. Ta procedura tworzy napisy XML dla produktów i typów produktów. Wykorzystuje ona metody z pakietu PL/SQL DBMS_XDB do usuwania i tworzenia plików zasobów XML w repozytorium Oracle XML DB Repository (jest to obszar składowania danych XML w bazie danych):

```
CREATE PROCEDURE create_xml_resources AS
  v_result BOOLEAN;

  -- tworzy napis zawierający XML dla produktów
  v_products VARCHAR2(300):=
  '<?xml version="1.0"?>' ||
  '<products>' ||
  '  <product product_id="1" product_type_id="1" name="Nauka współczesna" ' ||
  '    || price="19.95"/>' ||
  '  <product product_id="2" product_type_id="1" name="Chemia" ' ||
  '    || price="30"/>' ||
  '  <product product_id="3" product_type_id="2" name="Supernowa" ' ||
  '    || price="25.99"/>' ||
  '</products>';

  -- tworzy napis zawierający XML dla typów produktów
  v_product_types VARCHAR2(300):=
  '<?xml version="1.0"?>' ||
  '<product_types>' ||
  '  <product_type product_type_id="1" name="Książka"/>' ||
  '  <product_type product_type_id="2" name="Film"/>' ||
  '</product_types>';

BEGIN
  -- tworzy zasób dla produktów
  v_result := DBMS_XDB.CREATEROFILE('/public/products.xml',
  v_products);

  -- tworzy zasób dla typów produktów
  v_result := DBMS_XDB.CREATEROFILE('/public/product_types.xml',
```

```

    v_product_types);
END create_xml_resources;
/

```

Funkcja DBMS_XDB.CREATEROLESOURCE() tworzy zasób XML w bazie danych i zwraca wartość logiczną true lub false określającą, czy operacja zakończyła się powodzeniem. Dwa wywołania tej funkcji tworzą zasoby dla produktów i typów produktów w /public, będącej absolutną ścieżką do składowania zasobów.

Poniższa instrukcja wywołuje procedurę `create_xml_resources()`:

```
CALL create_xml_resources();
```

Poniższe zapytanie wykorzystuje funkcję `XMLQUERY()` do pobrania produktów z zasobu `/public/products.xml`:

```

SELECT XMLQUERY(
  'for $product in doc("/public/products.xml")/products/product
   return <product name="${$product/@name}" />'
  RETURNING CONTENT
)
AS xml_products
FROM DUAL;

XML_PRODUCTS
-----
<product name="Nauka współczesna"></product>
<product name="Chemia"></product>
<product name="Supernowa"></product>
```

Wyrażenie XQuery w funkcji `XMLQUERY()` w poprzednim przykładzie ma postać:

```
'for $product in doc("/public/products.xml")/products/product
 return <product name="${$product/@name}" />'
```

Omówmy kolejne składniki tego wyrażenia:

- Pętla `for` iteruje przez produkty w `/public/products.xml`.
- `$products` jest zmienną dowiązaną do sekwencji produktów zwróconych przez `doc("/public/products.xml")/products/product`, zwracające dokument `products.xml` składowany w `/public`. W każdej iteracji pętli zmiennej `$product` jest przypisywana kolejna wartość produktu z `products.xml`.
- Część `return` wyrażenia zwraca nazwę produktu z `$product`.

Kolejne zapytanie pobiera typy produktów z `/public/product_types.xml`:

```

SELECT XMLQUERY(
  'for $product_type in
   doc("/public/product_types.xml")/product_types/product_type
   return <product_type name="${$product_type/@name}" />'
  RETURNING CONTENT
)
AS xml_product_types
FROM DUAL;
```

```
XML_PRODUCT_TYPES
-----
<product_type name="Książka"></product_type>
<product_type name="Film"></product_type>
```

Poniższe zapytanie pobiera produkty, których cena jest większa niż 20 zł, a także typy tych produktów:

```

SELECT XMLQUERY(
  'for $product in doc("/public/products.xml")/products/product
  let $product_type :=
    doc("/public/product_types.xml")//product_type[@product_type_id =
      $product/@product_type_id]/@name
  where $product/@price > "20"
  order by $product/@product_id
  return <product name="${$product/@name}" />'
```

```

    product_type="{$product_type}" />
  RETURNING CONTENT
)
AS xml_query_results
FROM DUAL;

XML_QUERY_RESULTS
-----
<product name="Chemia" product_type="Książka"></product>
<product name="Supernowa" product_type="Film"></product>

```

Omówmy kolejne fragmenty wyrażenia XQuery w tym przykładzie:

- Wykorzystywane są dwie zmienne dowiązane: \$product i \$product_type. W tych zmiennych są przechowywane produkty i ich typy.
- Fragment `let` ustawia \$product_type na typ produktu pobranego z \$product. Wyrażenie po prawej stronie operatora `:=` wykonuje złączenie z wykorzystaniem wartości `product_type_id` składowanej w zmiennych \$product_type i \$product. `//` oznacza pobieranie wszystkich elementów.
- Część `where` pobiera jedynie produkty, których cena jest większa od 20.
- Część `order by` sortuje wyniki według identyfikatorów produktów (domyślnie w kolejności rosnącej).

W kolejnym przykładzie są używane następujące funkcje XQuery:

- `count()` zliczająca przesyłane do niej obiekty,
- `avg()` obliczająca średnią liczb przesyłanych do niej,
- `integer()` przycinająca liczbę i zwracającą liczbę całkowitą. Funkcja ta znajduje się w przestrzeni nazw xs. (Funkcje `mount()` i `avg()` znajdują się w przestrzeni nazw fn, do której baza danych odwołuje się automatycznie, co pozwala pominąć przestrzeń nazw przy wywoływaniu tych funkcji).

Poniższy przykład zwraca nazwę typu produktu, liczbę produktów każdego typu oraz średnią cenę produktów danego typu (przyciętą do liczby całkowitej):

```

SELECT XMLQUERY(
  'for $product_type in
  doc("/public/product_types.xml")/product_types/product_type
  let $product :=
    doc("/public/products.xml")//product[@product_type_id =
      $product_type/@product_type_id]
  return
    <product_type name="{$product_type/@name}"
      num_products="{$count($product)}"
      average_price="{$xs:integer(avg($product/@price))}">
    />'
  RETURNING CONTENT
)
AS xml_query_results
FROM DUAL;

```

```

XML_QUERY_RESULTS
-----
<product_type name="Książka" num_products="2" average_price="24">
</product_type>

<product_type name="Film" num_products="1" average_price="25">
</product_type>

```

Jak widać w wartościach w `num_products`, w bazie znajdują się dwie książki i jeden film. Średnia cena książek i filmów to odpowiednio 24 i 25, co widać w wartościach `average_price`.



Więcej informacji na temat funkcji można znaleźć pod adresem <http://www.w3.org/TR/x-query-operators>.

Zapisywanie XML w bazie danych

Z tego podrozdziału dowiesz się, jak składować dokumenty XML w bazie danych i pobierać informacje ze składowanych dokumentów XML.

Przykładowy plik XML

Będziemy używali pliku o nazwie *purchase_order.xml*, który jest plikiem XML zawierającym zamówienie. Ten plik znajduje się w katalogu *XML*.

Oto zawartość pliku *purchase_order.xml*:

```
<?xml version="1.0"?>
<purchase_order>
  <customer_order_id>176</customer_order_id>
  <order_date>2007-05-17</order_date>
  <customer_name>Świetne produkty</customer_name>
  <street>Wolności 10</street>
  <city>Sosnowiec</city>
  <state>SLA</state>
  <zip>94440</zip>
  <phone_number>555-121-1234</phone_number>
  <products>
    <product>
      <product_id>1</product_id>
      <name>Film Supernowa</name>
      <quantity>5</quantity>
    </product>
    <product>
      <product_id>2</product_id>
      <name>Książka Oracle SQL</name>
      <quantity>4</quantity>
    </product>
  </products>
</purchase_order>
```

W rzeczywistości zamówienie może zostać przesłane za pośrednictwem internetu do sklepu internetowego, który później je zrealizuje i wyśle towar do klienta.

Aby wykonywać równolegle prezentowane przykłady, należy skopiować katalog *XML* na partycję C serwera bazy danych. W systemie Linux lub Unix należy skopiować katalog na jedną z dostępnych partycji. Zanotuj położenie kopiwanych plików.

W kolejnych podrozdziałach nauczysz się, jak zapisać zawartość pliku XML do bazy danych.

Tworzenie przykładowego schematu XML

W katalogu *SQL* znajduje się skrypt SQL*Plus o nazwie *xml_schema.sql*. Tworzy on konto użytkownika o nazwie *xml_user* i hasło *xml_password*, a także wszystkie elementy używane w dalszej części rozdziału. Nie wykonuj jeszcze tego skryptu, ponieważ prawdopodobnie trzeba go wcześniej wyedytować. Najpierw opiszę skrypt, a następnie wskażę, co należy zmodyfikować.

Skrypt zawiera instrukcje tworzące następujące elementy w bazie danych:

- typ obiektowy o nazwie *t_product* (reprezentujący produkty),
- tabelę zagnieżdzoną *t_nested_table_product* (reprezentuje tabelę zagnieżdzoną produktów)
- tabelę o nazwie *purchase_order*.

Poniższy przykład pokazuje instrukcje z pliku *xml_schema.sql* tworzące elementy opisane na powyższej liście:

```
CREATE TYPE t_product AS OBJECT (
  product_id INTEGER,
  name VARCHAR2(15),
```

```

quantity INTEGER
);
/
CREATE TYPE t_nested_table_product AS TABLE OF t_product;
/
CREATE TABLE purchase_order (
    purchase_order_id INTEGER CONSTRAINT purchase_order_pk PRIMARY KEY,
    customer_order_id INTEGER,
    order_date DATE,
    customer_name VARCHAR2(25),
    street VARCHAR2(15),
    city VARCHAR2(15),
    state VARCHAR2(3),
    zip VARCHAR2(5),
    phone_number VARCHAR2(12),
    products t_nested_table_product,
    xml_purchase_order XMLType
)
NESTED TABLE products
STORE AS nested_products;

```

Należy zauważyc, że kolumna `xml_purchase_order` jest typu `XMLType`, który jest typem wbudowanym do bazy danych Oracle i umożliwia składowanie danych XML.

Skrypt `xml_schema.sql` zawiera również poniższą instrukcję, która tworzy obiekt katalogu o nazwie `XML_FILES_DIR`:

```
CREATE OR REPLACE DIRECTORY XML_FILES_DIR AS 'C:\XML';
```

Jeżeli katalog `XML` został skopiowany w inne miejsce niż główny katalog partycji C albo jeśli używasz systemu Linux lub Unix, należy zmodyfikować ten wiersz. Jeżeli jest to konieczne, należy to zrobić od razu i zapisać skrypt.

Poniższa instrukcja `INSERT` (również zamieszczona w skrypcie) wstawia wiersz do tabeli `purchase_order`:

```

INSERT INTO purchase_order (
    purchase_order_id,
    xml_purchase_order
) VALUES (
    1,
    XMLType(
        BFILENAME('XML_FILES_DIR', 'purchase_order.xml'),
        NLS_CHARSET_ID('AL32UTF8')
    )
);

```

Jak widać, konstruktor `XMLType()` przyjmuje dwa parametry. Pierwszym z nich jest `BFILE`, będący wskaźnikiem do zewnętrznego pliku. Drugi parametr określa zestaw znaków tekstu w pliku zewnętrznym. W przedstawionej instrukcji obiekt `BFILE` wskazuje na plik `purchase_order.xml`, zestaw znaków jest ustawiony na `AL32UTF8`, czyli standardowe kodowanie UTF-8. Po uruchomieniu instrukcji `INSERT` plik jest odczytywany, a następnie składowany w bazie danych w kolumnie `xml_purchase_order` tabeli `purchase_order`.



Szczegółowe informacje na temat różnych zestawów znaków można znaleźć w podręczniku *Oracle Globalization Support Guide* opublikowanym przez Oracle Corporation.

Możemy zauważyc, że kolumny `customer_order_id`, `order_date`, `customer_name`, `street`, `city`, `state`, `zip`, `phone_number` i `products` w tabeli `purchase_order` są puste. Dane dla tych kolumn mogą zostać wydobycie z XML składowanego w kolumnie `xml_purchase_order`. W dalszej części tego rozdziału zostanie przedstawiona procedura PL/SQL odczytująca XML i ustawiająca odpowiednie wartości kolumn.

Teraz można uruchomić skrypt `xml_schema.sql` jako użytkownik z podwyższonymi uprawnieniami (na przykład `system`):

```
CONNECT system/oracle
@ C:\sql_book\SQL\xml_schema.sql
```

Po zakończeniu skryptu będzie zalogowany użytkownik `xml_user`.

Pobieranie informacji z przykładowego schematu XML

Z tego podrozdziału dowiesz się, jak pobrać informacje ze schematu `xml_user`.

Poniższy przykład pobiera wiersz z tabeli `purchase_order`:

```
SET LONG 1000
SET PAGESIZE 500
SELECT purchase_order_id, xml_purchase_order
FROM purchase_order;

PURCHASE_ORDER_ID
-----
XML_PURCHASE_ORDER
-----
1
<?xml version="1.0"?>
<purchase_order>
<customer_order_id>176</customer_order_id>
<order_date>2007-05-17</order_date>
<customer_name>Świetne produkty</customer_name>
<street>Wolności 10</street>
<city>Sosnowiec</city>
<state>SLA</state>
<zip>94440</zip>
<phone_number>555-121-1234</phone_number>
<products>
<product>
<product_id>1</product_id>
<name>Film Supernowa</name>
<quantity>5</quantity>
</product>
<product>
<product_id>2</product_id>
<name>Książka Oracle SQL</name>
<quantity>4</quantity>
</product>
</products>
</purchase_order>
```

Kolejne zapytanie wydobywa za pomocą funkcji `EXTRACT()` elementy `customer_order_id`, `order_date`, `customer_name` i `phone_number` z XML składowanego w kolumnie `xml_purchase_order`:

```
SELECT
  EXTRACT(xml_purchase_order,
    '/purchase_order/customer_order_id') cust_order_id,
  EXTRACT(xml_purchase_order, '/purchase_order/order_date') order_date,
  EXTRACT(xml_purchase_order, '/purchase_order/customer_name') cust_name,
  EXTRACT(xml_purchase_order, '/purchase_order/phone_number') phone_number
FROM purchase_order
WHERE purchase_order_id = 1;

CUST_ORDER_ID
-----
ORDER_DATE
-----
CUST_NAME
-----
PHONE_NUMBER
-----
```

```
<customer_order_id>176</customer_order_id>
<order_date>2007-05-17</order_date>
<customer_name>Świetne produkty</customer_name>
<phone_number>555-121-1234</phone_number>
```

Funkcja EXTRACT() zwraca wartości jako obiekty XMLType.

Do pobrania wartości jako napisów służy funkcja EXTRACTVALUE(). Na przykład poniższe zapytanie wydobywa te same wartości jako napisy, korzystając z funkcji EXTRACTVALUE():

```
SELECT
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/customer_order_id') cust_order_id,
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/order_date') order_date,
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/customer_name') cust_name,
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/phone_number') phone_number
FROM purchase_order
WHERE purchase_order_id = 1;
```

```
CUST_ORDER_ID
-----
176

ORDER_DATE
-----
2007-05-17

CUST_NAME
-----
Świetne produkty

PHONE_NUMBER
-----
555-121-1234
```

Następne zapytanie wydobywa i konwertuje za pomocą funkcji TO_DATE wartość order_date na typ DATE. Należy zauważyć, że format, w jakim data jest składowana w XML, został przesłany jako drugi parametr funkcji TO_DATE(). Zwraca ona datę w domyślnym formacie używanym przez bazę danych (YY/MM/DD):

```
SELECT
  TO_DATE(
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/order_date'),
    'YYYY-MM-DD'
  ) AS ord_date
FROM purchase_order
WHERE purchase_order_id = 1;
```

```
ORD_DATE
-----
07/05/17
```

Poniższe zapytanie pobiera za pomocą funkcji EXTRACT() wszystkie produkty z tabeli xml_purchase_order jako XML. Należy zauważyć użycie // do pobrania wszystkich wierszy:

```
SELECT
  EXTRACT(xml_purchase_order, '/purchase_order//products') xml_products
FROM purchase_order
WHERE purchase_order_id = 1;
```

```
XML_PRODUCTS
-----
<products>
  <product>
    <product_id>1</product_id>
    <name>Film Supernowa</name>
    <quantity>5</quantity>
  </product>
```

510 Oracle Database 12c i SQL. Programowanie

```
<product>
  <product_id>2</product_id>
  <name>Książka Oracle SQL</name>
  <quantity>4</quantity>
</product>
</products>
```

Następne zapytanie pobiera produkt nr 2 z tabeli `xml_purchase_order`. Należy zauważyć, że `product[2]` zwraca produkt nr 2:

```
SELECT
  EXTRACT(
    xml_purchase_order,
    '/purchase_order/products/product[2]'
  ) xml_product
FROM purchase_order
WHERE purchase_order_id = 1;
```

`XML_PRODUCT`

```
<product>
  <product_id>2</product_id>
  <name>Książka Oracle SQL</name>
  <quantity>4</quantity>
</product>
```

Poniższe zapytanie pobiera z tabeli `xml_purchase_order` produkt „Film Supernowa”. Należy zauważać, że nazwa produktu do pobrania została umieszczona w nawiasach kwadratowych:

```
SELECT
  EXTRACT(
    xml_purchase_order,
    '/purchase_order/products/product[name="Film Supernowa"]'
  ) xml_product
FROM purchase_order
WHERE purchase_order_id = 1;
```

`XML_PRODUCT`

```
<product>
  <product_id>1</product_id>
  <name>Film Supernowa</name>
  <quantity>5</quantity>
</product>
```

Funkcja `EXISTSNODE()` sprawdza, czy element XML istnieje. Jeżeli istnieje, funkcja `EXISTSNODE()` zwraca 1; w przeciwnym razie zwraca 0. Na przykład poniższe zapytanie zwraca napis 'Istnieje', ponieważ produkt nr 1 istnieje w tabeli:

```
SELECT 'Istnieje' AS "EXISTS"
FROM purchase_order
WHERE purchase_order_id = 1
AND EXISTSNODE(
  xml_purchase_order,
  '/purchase_order/products/product[product_id=1]'
) = 1;
```

`EXISTS`

Istnieje

Kolejne zapytanie nie zwraca wierszy, ponieważ nie istnieje produkt nr 3:

```
SELECT 'Istnieje'
FROM purchase_order
WHERE purchase_order_id = 1
```

```

AND EXISTSNODE(
    xml_purchase_order,
    '/purchase_order/products/product [product_id=3]'
) = 1;

no rows selected

```

Następne zapytanie pobiera produkty jako tablicę o zmiennej długości zawierającą obiekty XMLType, korzystając z funkcji XMLSEQUENCE(). Należy zauważyć, że do pobrania wszystkich produktów i ich elementów użyto product.*:

```

SELECT product.*
FROM TABLE(
    SELECT
        XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
    FROM purchase_order
    WHERE purchase_order_id = 1
) product;

COLUMN_VALUE
-----
<product>
<product_id>1</product_id>
<name>Film Supernowa</name>
<quantity>5</quantity>
</product>

<product>
<product_id>2</product_id>
<name>Książka Oracle SQL</name>
<quantity>4</quantity>
</product>

```

Następne zapytanie pobiera jako napisy product_id, name i quantity produktów, korzystając z funkcji EXTRACTVALUE():

```

SELECT
    EXTRACTVALUE(product.COLUMN_VALUE, '/product/product_id') AS product_id,
    EXTRACTVALUE(product.COLUMN_VALUE, '/product/name') AS name,
    EXTRACTVALUE(product.COLUMN_VALUE, '/product/quantity') AS quantity
FROM TABLE(
    SELECT
        XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
    FROM purchase_order
    WHERE purchase_order_id = 1
) product;

PRODUCT_ID
-----
NAME
-----
QUANTITY
-----
1
Film Supernowa
5

2
Książka Oracle SQL
4

```

Aktualizowanie informacji w przykładowym schemacie XML

W tabeli purchase_order kolumny customer_order_id, order_date, customer_name, street, city, state, zip, phone_number i products są puste. Dane dla tych kolumn mogą zostać wydobyte z XML składowanego w kolumnie xml_purchase_order.

W tym podrozdziale zostanie przedstawiona procedura o nazwie `update_purchase_order()`, która odczytuje XML i odpowiednio ustawia wartości w pozostałych kolumnach.

Najbardziej złożonym aspektem procedury `update_purchase_order()` jest proces odczytywania produktów z XML i zapisywania ich w kolumnie `products` tabeli `purchase_order`. Proces ten jest realizowany przez procedurę w kilku krokach:

1. Kursor odczytuje produkty z XML składowanego w kolumnie `xml_purchase_order`.
2. XML jest konwertowany na napisy za pomocą funkcji `EXTRACTVALUE()`.
3. Uzyskane napisy są zapisywane w kolumnie `products`, która jest tabelą zagnieżdżoną. Tabela zagnieżdżona przechowuje wiele produktów zamówienia.

Poniższa instrukcja (zamieszczona w skrypcie `xml_schema.sql`) tworzy procedurę `update_purchase_order()`:

```

CREATE PROCEDURE update_purchase_order(
    p_purchase_order_id IN purchase_order.purchase_order_id%TYPE
) AS
    v_count INTEGER := 1;

    -- deklaracja tabeli zagnieżdzonej, w której będą składowane produkty
    v_nested_table_products t_nested_table_product :=
        t_nested_table_product();

    -- deklaracja typu reprezentującego rekord produktu
    TYPE t_product_record IS RECORD (
        product_id INTEGER,
        name VARCHAR2(45),
        quantity INTEGER
    );

    -- deklaracja typu REF CURSOR wskazującego na rekordy produktów
    TYPE t_product_cursor IS REF CURSOR RETURN t_product_record;

    -- deklaracja kursora
    v_product_cursor t_product_cursor;

    -- deklaracja zmiennej przechowującej rekord produktu
    v_product t_product_record;
BEGIN
    -- otwiera v_product_cursor do odczytania product_id, name, and quantity
    -- każdego produktu składowanego w XML z kolumny xml_purchase_order
    -- tabeli purchase_order table
    OPEN v_product_cursor FOR
        SELECT
            EXTRACTVALUE(product.COLUMN_VALUE, '/product/product_id')
            AS product_id,
            EXTRACTVALUE(product.COLUMN_VALUE, '/product/name') AS name,
            EXTRACTVALUE(product.COLUMN_VALUE, '/product/quantity') AS quantity
        FROM TABLE(
            SELECT
                XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
            FROM purchase_order
            WHERE purchase_order_id = p_purchase_order_id
        ) product;

    -- pętla przez zawartość v_product_cursor
    LOOP
        -- pobiera rekordy produktów z v_product_cursor
        -- przerwia, gdy nie ma więcej rekordów
        FETCH v_product_cursor INTO v_product;
        EXIT WHEN v_product_cursor%NOTFOUND;
    END LOOP;

```

```

-- rozszerza tabelę v_nested_table_products,
-- aby można było w niej zapisać produkt
v_nested_table_products.EXTEND;

-- tworzy nowy produkt i zapisuje go w w v_nested_table_products
v_nested_table_products(v_count) :=
  t_product(v_product.product_id, v_product.name, v_product.quantity);

-- wyświetla nowy produkt zapisany w v_nested_table_products
DBMS_OUTPUT.PUT_LINE('product_id = ' ||
  v_nested_table_products(v_count).product_id);
DBMS_OUTPUT.PUT_LINE('name = ' ||
  v_nested_table_products(v_count).name);
DBMS_OUTPUT.PUT_LINE('quantity = ' ||
  v_nested_table_products(v_count).quantity);

-- zwiększa v_count, przygotowując kolejną iterację pętli
v_count := v_count + 1;
END LOOP;

-- zamyka v_product_cursor
CLOSE v_product_cursor;

-- modyfikuje tabelę purchase_order, używając
-- wartości wydobytych z XML składowanego w kolumnie
-- xml_purchase_order (zagnieżdżona tabela products,
-- jest ustawiana na obiekt v_nested_table_products,
-- w którym znajdują się dane z poprzedniej pętli)
UPDATE purchase_order
SET
  customer_order_id =
    EXTRACTVALUE(xml_purchase_order,
      '/purchase_order/customer_order_id'),
  order_date =
    TO_DATE(EXTRACTVALUE(xml_purchase_order,
      '/purchase_order/order_date'), 'YYYY-MM-DD'),
  customer_name =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/customer_name'),
  street =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/street'),
  city =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/city'),
  state =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/state'),
  zip =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/zip'),
  phone_number =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/phone_number'),
  products = v_nested_table_products
WHERE purchase_order_id = p_purchase_order_id;

-- zatwierdza transakcję
COMMIT;
END update_purchase_order;
/

```

W poniższym przykładzie włączono wypisywanie z serwera i wywołano procedurę `update_purchase_order` w celu modyfikacji zamówienia nr 1:

```

SET SERVEROUTPUT ON
CALL update_purchase_order(1);
product_id = 1
name = Film Supernowa
quantity = 5

```

```
product_id = 2
name = Książka Oracle SQL
quantity = 4
```

Poniższe zapytanie pobiera kolumny zamówienia nr 1:

```
SELECT purchase_order_id, customer_order_id, order_date, customer_name,
       street, city, state, zip, phone_number, products
  FROM purchase_order
 WHERE purchase_order_id = 1;
```

```
PURCHASE_ORDER_ID CUSTOMER_ORDER_ID ORDER_DATE CUSTOMER_NAME
-----
STREET          CITY           STATE ZIP PHONE_NUMBER
-----
PRODUCTS(PRODUCT_ID, NAME, QUANTITY)
-----
      1           176 07/05/17 Świecone produkty
Wolność 10     Sosnowiec      SLA 94440 555-121-1234
T_NESTED_TABLE_PRODUCT(
    T_PRODUCT(1, 'Film Supernowa', 5),
    T_PRODUCT(2, 'Książka Oracle SQL', 4)
)
```

Tabela zagnieżdżona `products` zawiera takie same dane jak te składowane w elementach XML produktu, składowanych z kolei w kolumnie `xml_purchase_order`. Powyżej dodano kilka podziałów wierszy w celu oddzielenia produktów w wynikach, aby zwiększyć ich czytelność.

Podsumowanie

Z tego rozdziału dowiedziałeś się:

- jak generować kod XML z danych relacyjnych,
- jak zapisywać XML do bazy danych.

Więcej informacji na temat XML można znaleźć w podręcznikach *Oracle XML Developer's Kit* i *Oracle XML DB Developer's Guide* opublikowanych przez Oracle Corporation.

To już jest w zasadzie koniec książki (pozostał jeszcze dodatek A). Mam nadzieję, że była ona pouczająca i przydatna, a także interesująca!

DODATEK

A

Typy danych Oracle

Ten dodatek opisuje główne typy danych Oracle SQL i PL/SQL.

Typy w Oracle SQL

W tabeli A.1 opisano typy w Oracle SQL.

Tabela A.1. Typy w Oracle SQL

Typ	Opis
CHAR[(<i>długość</i> [BYTE CHAR])] ¹	Dane znakowe o stałej długości wynoszącej <i>długość</i> znaków lub bajtów. Dane są dopełniane spacjami kończącymi. Maksymalna długość wynosi 2 000 bajtów
VARCHAR2[(<i>długość</i> [BYTE CHAR])] ¹	Dane znakowe o zmiennej długości wynoszącej do <i>długość</i> znaków lub bajtów. Maksymalna długość wynosi 32 767 bajtów
NCHAR[<i>(długość)</i>]	Dane znakowe Unicode o stałej długości wynoszącej <i>długość</i> znaków. Liczba składowanych bajtów jest iloczynem 2 i <i>długość</i> w przypadku kodowania AL16UTF16 lub iloczynem 3 i <i>długość</i> w przypadku kodowania UTF-8. Maksymalna długość wynosi 2 000 bajtów
NVARCHAR2(<i>długość</i>)	Dane znakowe Unicode o zmiennej długości do <i>długość</i> znaków. Liczba składowanych bajtów jest iloczynem 2 i <i>długość</i> w przypadku kodowania AL16UTF16 lub iloczynem 3 i <i>długość</i> w przypadku kodowania UTF-8. Maksymalna długość wynosi 32 767 bajtów
BINARY_FLOAT	Typ wprowadzony w Oracle Database 10g przechowuje 32-bitową liczbę zmiennoprzecinkową o pojedynczej precyzji. Operacje na typach BINARY_FLOAT są zwykle wykonywane szybciej niż na wartościach NUMBER.
BINARY_DOUBLE	Typ wprowadzony w Oracle Database 10g przechowuje 64-bitową liczbę zmiennoprzecinkową o podwójnej precyzji. Operacje na typach BINARY_DOUBLE są zwykle wykonywane szybciej niż na wartościach NUMBER.
NUMBER(<i>precyzja, skala</i>) i NUMERIC(<i>precyzja, skala</i>)	Liczba o zmiennej długości. <i>precyzja</i> określa maksymalną liczbę cyfr (po lewej i prawej stronie separatora dziesiętnego, jeżeli jest używany). Maksymalna obsługiwana precyzja wynosi 38.

Tabela A.1. Typy w Oracle SQL — ciąg dalszy

Typ	Opis
NUMBER(<i>precyzja, skala</i>) i NUMERIC(<i>precyzja, skala</i>)	<i>skala</i> określa maksymalną liczbę cyfr po prawej stronie separatora dziesiętnego (jeżeli jest używany). <i>skala</i> może mieć wartość w zakresie od -84 do 127. Jeżeli parametry <i>precyzja</i> i <i>skala</i> nie zostaną określone, liczby będą składowane z precyzją i skalą wynoszącymi 38, co oznacza, że można przesłać liczbę złożoną z 38 cyfr, z których każda może znajdować się po lewej lub prawej stronie separatora dziesiętnego
DEC i DECIMAL	Podtyp NUMBER. Liczba dziesiętna stałoprzecinkowa z maksymalnie 38-cyfrową precyzją dziesiętną
DOUBLE PRECISION i FLOAT	Podtyp NUMBER. Liczba zmiennoprzecinkowa z maksymalnie 38-cyfrową precyzją dziesiętną
REAL	Podtyp NUMBER. Liczba zmiennoprzecinkowa z maksymalnie 18-cyfrową precyzją dziesiętną
INT, INTEGER i SMALLINT	Podtyp NUMBER. Liczba całkowita z maksymalnie 38-cyfrową precyzją dziesiętną
DATE	Data, czas i wiek. Zawiera wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), liczbę minut i sekund. Może przechowywać daty między 1 stycznia 4712 p.n.e. i 31 grudnia 9999 n.e.
INTERVAL YEAR[(<i>precyzja_lat</i>)] TO MONTH	Interwał czasu mierzony w latach i miesiącach. Parametr <i>precyzja_lat</i> określa precyzję lat i może przyjmować wartości od 0 do 9 (domyślnie 2). Ten typ może reprezentować zarówno interwał dodatni, jak i ujemny
INTERVAL DAY[(<i>precyzja_dni</i>)] TO SECOND[(<i>precyzja_sekund</i>)]	Interwał czasu mierzony w dniach i sekundach. Parametr <i>precyzja_dni</i> określa precyzję dni i może przyjmować wartości od 0 do 9 (domyślnie 2). Parametr <i>precyzja_sekund</i> określa precyzję sekund i może przyjmować wartości od 0 do 9 (domyślnie 6). Ten typ może reprezentować zarówno interwał dodatni, jak i ujemny
TIMESTAMP[<i>(precyzja_sekund)</i>] WITH TIME ZONE	Data, czas oraz wiek. Zawiera wszystkie cztery cyfry roku, miesiąc, dzień, godzinę (w formacie 24-godzinnym), liczbę minut i sekund. Parametr <i>precyzja_sekund</i> określa precyzję sekund i może przyjmować wartości od 0 do 9 (domyślnie 6). Rozszerza typ TIMESTAMP o przechowywanie strefy czasowej. Strefa czasowa może być określona jako przesunięcie względem UTC (na przykład -8:0) lub przez nazwę regionu, na przykład US/Pacific lub PST.
TIMESTAMP[<i>(precyzja_sekund)</i>] WITH LOCAL TIME ZONE	Rozszerza typ TIMESTAMP o konwersję przesłanej daty i czasu na lokalną strefę czasową ustaloną w bazie danych. Proces konwersji nazywamy normalizacją.
CLOB	Dane znakowe o zmiennej długości i pojedynczych bajtach o wielkości do 128 terabajtów (zależne od rozmiaru bloku w bazie danych)
NCLOB	Dane o zmiennej długości w zestawie znaków narodowych Unicode o wielkości do 128 terabajtów (zależne od rozmiaru bloku w bazie danych)
BLOB	Dane binarne o zmiennej długości o wielkości do 128 terabajtów (zależne od rozmiaru bloku w bazie danych)
BFILE	Wskaźnik do pliku zewnętrznego. Plik zewnętrzny nie jest składowany w bazie danych. Obsługiwane są pliki o maksymalnej wielkości 4 gigabajty

Tabela A.1. Typy w Oracle SQL — ciąg dalszy

Typ	Opis
LONG	Dane znakowe o zmiennej długości (do 2 gigabajtów). Jest wypierany przez typy CLOB i NCLOB, ale wciąż obsługiwany w celu zachowania kompatybilności wstępnej
RAW(<i>długość</i>)	Dane binarne o zmiennej długości do <i>długość</i> bajtów. Maksymalna długość wynosi 32 767 bajtów. Jest wypierany przez typ BLOB, ale wciąż obsługiwany w celu zachowania kompatybilności wstępnej
LONG RAW	Dane binarne o zmiennej długości do 2 gigabajtów. Jest wypierany przez typ BLOB, ale wciąż obsługiwany w celu zachowania kompatybilności wstępnej
ROWID	Napis w kodowaniu base-64 reprezentujący adres wiersza
UROWID[<i>(długość)</i>]	Napis w kodowaniu base-64 reprezentujący logiczny adres wiersza w indeksowanej tabeli.
	Parametr <i>długość</i> określa liczbę bajtów. Maksymalna długość wynosi 4 000 bajtów (jest to też wartość domyślna)
REF <i>typ obiektowy</i>	Odwołanie do typu obiektowego. Przypomina wskaźnik z języka C++
VARRAY	Tablica o zmiennej długości. Jest to typ złożony, w którym jest składowany uporządkowany zbiór elementów
NESTED TABLE	Tabela zagnieżdziona. Jest to typ złożony, w którym jest składowany nieuporządkowany zbiór elementów
XMLType	Przechowuje dane XML
Typ obiektowy zdefiniowany przez użytkownika	Można definiować własne typy obiektowe i tworzyć obiekty tych typów. Ten temat został opisany w rozdziale 13.

¹ Słowa kluczowe BYTE i CHAR są obsługiwane jedynie przez Oracle Database 9i i nowsze. Jeżeli nie zostanie określone żadne z tych słów, domyślnie zostanie przyjęte BYTE.

Typy w Oracle PL/SQL

Oracle PL/SQL obsługuje wszystkie typy opisane w tabeli A.1, a także typy dodatkowe, opisane w tabeli A.2.

Tabela A.2. Typy w Oracle PL/SQL

Typ	Opis
BOOLEAN	Zmienna logiczna (TRUE, FALSE lub NULL)
BINARY_INTEGER	Liczba całkowita z zakresu od -2 147 483 647 do 2 147 483 647
NATURAL	Podtyp BINARY_INTEGER. Nieujemna liczba całkowita z zakresu od 0 do 2 147 483 647
NATURALN	Podtyp BINARY_INTEGER. Nieujemna liczba całkowita z zakresu od 0 do 2 147 483 647 (nie może mieć wartości NULL)
POSITIVE	Podtyp BINARY_INTEGER. Dodatnia liczba całkowita z zakresu od 1 do 2 147 483 647
POSITIVEN	Podtyp BINARY_INTEGER. Dodatnia liczba całkowita z zakresu od 1 do 2 147 483 647 (nie może mieć wartości NULL)
SIGNTYPE	Podtyp BINARY_INTEGER. Liczba całkowita mająca wartość -1, 0 lub 1
PLS_INTEGER	To samo co BINARY_INTEGER
SIMPLE_INTEGER	Nowość w Oracle Database 11g. Typ SIMPLE_INTEGER jest podtypem BINARY_INTEGER i może przechowywać wartości z tego samego przedziału, oprócz wartości NULL. Ponadto przepelnienie arytmetyczne nie powoduje wystąpienia błędu, jeżeli jest używany typ SIMPLE_INTEGER. Zamiast tego wynik jest po prostu przycinany

Tabela A.2. Typy w Oracle PL/SQL — ciąg dalszy

Typ	Opis
STRING	To samo co VARCHAR2
RECORD	Złożona grupa innych typów. Przypomina strukturę z języka C++
REF CURSOR	Wskaźnik do zestawu wierszy

Skorowidz

A

ACID, 254
aktualizowanie informacji, 511
aktywacja ról, 276
aliasy
 kolumn, 48
 tabel, 59
analiza danych, 205
ANSI, 59, 417
archiwa migawek, 314
atrybut :new, 464
Automatic Database Diagnostic Monitor, 488
automatyczne generowanie instrukcji, 88

B

baza danych
 sklepu, 30
 store, 33
B-drzewo, 302
bloki, 317
blokowanie transakcji, 255
błędy, 160
błędy w procedurze, 335

C

cudzysłowy, 245
czas, 127
czyszczanie formatowania, 78

D

dane LOB, 426
data, 46, 127
Database Replay, 488
datowniki, 143
DCL, Data Control Language, 24
DDL, Data Definition Language, 24
deaktywacja ról, 276
definiowanie
 kolumny, 389
 konstruktora, 379
 zmiennych, 81
DML, Data Manipulation Language, 24
dodawanie
 indeksów, 474
 kommentarza, 291
 wierszy, 38
wierzów
 CHECK, 286
 FOREIGN KEY, 287
 NOT NULL, 287
 UNIQUE, 288
dołączane bazy danych, 31
domyślna wartość kolumny, 300
domyślny formatu daty, 134
dostęp do
 danych LOB, 426
 komórek, 224, 227
 zakresu komórek, 225
drzewo, 181, 185
duże obiekty, LOB, 425, 427
działania arytmetyczne, 45

dziedziczenie
 atrybutów, 369, 384
 typów, 368

E

EDIT, 73
edykcja instrukcji, 72
edytor, 75
elementy w kolekcji, 392
eliminowanie
 gałęzi, 185
 węzłów, 185

F

filtrowanie wierszy, 124, 199, 471
formacja
 typu V, 238, 241
 typu W, 240
format
 daty i czasu, 133
 RR, 136
 YY, 135
formatowanie
 kolumn, 76, 78
 liczb, 110
 wyników, 183
funkcja, 335
 ABS(), 100
 ADD_MONTHS(), 138
 ASCII(), 93
 ASCIISTR(), 105
 AVG(), 118
 BIN_TO_NUM(), 105
 CARDINALITY(), 420

funkcja		
CAST(), 105, 399	NTILE(), 211	TREAT(), 375
CEIL(), 100	NUMTODSINTERVAL(), 155	TRIM(), 96
CHARTOROWID(), 106	NUMTOYMINTERVAL(), 155	TRUNC(), 103, 140
CHR(), 93	NVL(), 50, 96	UNISTR(), 112
COLLECT(), 422	NVL2(), 97	UPPER(), 95
COMPOSE(), 107	PERCENT_RANK(), 211	VARIANCE(), 120
CONCAT(), 93	POWER(), 101	XMLAGG(), 495
CONVERT(), 107	POWERMULTISET(), 423	XMLATTRIBUTES(), 494
COUNT(), 119	POWERMULTISET_BY_CARDINALITY(), 423	XMLCOLATTVAL(), 497
CUME_DIST(), 211	PRESENTNNV(), 229	XMLCOMMENT(), 499
CURRENT_TIMESTAMP, 147	PRESENTV(), 228	XMLCONCAT(), 498
CURRENTV(), 226	RANK(), 206	XMLEMENT(), 492
DECODE(), 177	RATIO_TO_REPORT(), 219	XMLFOREST(), 494
DECOMPOSE(), 107	RAWTOHEX(), 108	XMLPARSE(), 498
DENSE_RANK(), 206	REGEXP_COUNT(), 117	XMLPI(), 499
EXTRACT(), 148	REGEXP_INSTR(), 116	XMLQUERY(), 502
FIRST(), 205, 221	REGEXP_LIKE(), 114	XMLSEQUENCE(), 500
FIRST_VALUE(), 216	REGEXP_REPLACE(), 117	XMLSERIALIZE(), 501
FLOOR(), 100	REGEXP_SUBSTR(), 117	funkcje
FROM_TZ(), 149	REPLACE(), 97	agregujące, 117, 232
get_product(), 364	ROUND(), 102, 139	analityczne, 205
get_product_ref(), 365	ROW_NUMBER(), 211	hipotetycznego rankingu i rozkładu, 222
get_products(), 361	ROWIDTOCHAR(), 108	hipotetycznych klasyfikacji i dystrybucji, 205
GREATEST(), 101	RPAD(), 96	jednowierszowe, 91
GROUP_ID(), 200	RTRIM(), 96	klasyfikujące, 205, 207
GROUPING(), 195, 196	SET(), 421	konwertujące, 103, 128
GROUPING_ID(), 198, 199	SIGN(), 102	numeryczne, 98
HEXTORAW(), 107	SOUNDEX(), 97	obiektów, 372
INITCAP(), 94	SQRT(), 102	okien, 205
INSTR(), 94	STDDEV(), 120	operujące na
IS OF(), 372	SUBSTR(), 97	datach i godzinach, 137
LAG(), 205, 220	SUM(), 120	datownikach, 147
LAST(), 205, 221	SYS_EXTRACT_UTC(), 149	interwałach, 155
LAST_DAY(), 138	SYS_TYPEID(), 378	strefach czasowych, 141
LAST_VALUE(), 216	SYSDATE, 140	wyrażeniach regularnych, 115
LEAD(), 205, 220	SYSTIMESTAMP, 147	rankingowe, 212
LEAST(), 101	TABLE(), 394	raportujące, 205, 218
LENGTH(), 95	TO_BINARY_DOUBLE(), 108	regresji liniowej, 205, 221
LISTAGG(), 220	TO_BINARY_FLOAT(), 108	wyrażeń regularnych, 112
LOCALTIMESTAMP, 147	TO_CHAR(), 108, 128, 134	znakowe, 92
LOWER(), 95	TO_DATE(), 46, 128, 132, 509	
LPAD(), 96	TO_MULTI_BYTE(), 109	
LTRIM(), 96	TO_NUMBER(), 110	
MAX(), 119	TO_SINGLE_BYTE(), 111	
MIN(), 119	TO_TIMESTAMP(), 150	
MONTHS_BETWEEN(), 138	TO_TIMESTAMP_TZ(), 150	
NEXT_DAY(), 139	TRANSLATE(), 176	
NTH_VALUE(), 217		

G

generowanie
instrukcji, 88
natynego kodu
maszynowego, 347

planu wykonania, 483
XML, 492
grupowanie wierszy, 120
gwiazdka, 35

H

hurtownia danych, 305

I

identyfikatory
 obiektów, 357
 wierszy, 44
iloczyn kartezjański, 60
indeks, 302
 bitmapowy, 305, 475
 oparty na funkcji, 303
 typu B-drzewo, 303, 475
 złożony, 303
informacje o
 definicjach perspektyw, 310
 dużych obiektach, 425
 funkcjach, 337, 340
 indeksach, 304
 indeksach kolumny, 304
 kolekcjach, 387, 389
 kolumnach, 283
 optymalizacji, 471
 procedurach, 334, 340
 sekwencjach, 298
 tabelach, 282
 tabeli customers, 29
 tabeli zagnieżdzonej, 390
typach obiektowych, 351
więzach, 289, 290
więzach perspektywy, 311
wyzwalaczach, 343

instrukcja
 ALTER ROLE, 276
 ALTER TABLE, 287, 315
 CREATE FLASHBACK
 ARCHIVE, 314
 CREATE PROCEDURE, 41
 DELETE, 168, 246
 DESCRIBE, 295
 DISCONNECT, 40
 DROP ROLE, 277
 DROP TABLE, 88

INSERT, 39, 243, 308
MERGE, 249
OPEN-FOR, 325
REVOKE, 276
ROLLBACK, 251
SELECT, 26, 39, 43
UPDATE, 167, 245

instrukcje

 DCL, 24
 DDL, 24, 32
 DML, 24
 identyczne, 478
 nieidentyczne, 478
 TC, 24

integralność bazy danych, 247

interwały czasowe, 127, 151

 DAY TO SECOND, 154
 YEAR TO MONTH, 152

izolacja transakcji, 256

J

język
 Perl, 114
 PL/SQL, 41, 317
 SQL, 24
 XML, 491

K

klauzula
 CONNECT BY, 182
 CROSS APPLY, 201
 CUBE, 190, 194, 478
 DEFAULT ON NULL, 293
 FETCH FIRST, 234
 FROM, 44, 160
 GROUP BY, 120, 124, 199
 GROUPING SETS, 197, 478
 HAVING, 124, 159, 199, 475
 LATERAL, 202
 MATCH_RECOGNIZE,
 237–239
 MODEL, 223, 227–229
 NOT FINAL, 368
 OFFSET, 235
 ORDER BY, 57, 161
 OUTER APPLY, 201, 202
 PARTITION BY, 208

PERCENT, 236
PIVOT, 230
RETURNING, 246
ROLLUP, 190–193
START WITH, 182–185
UNION, 476
UNION ALL, 476
UNPIVOT, 230, 233
WHERE, 44, 124, 157, 471
WITH, 347
 WITH TIES, 236

klucz

 główny, 34
 główny złożony, 37
 obcy, 35

kolejność wykonywania działań,
 48

kolekcje, 387

kolekcje wielopoziomowe, 411,
 412

kolumny, 23

 niewidoczne, 294, 313
 typu IDENTITY, 301
 widoczne, 313
 wirtualne, 284

komentarze, 291

kompresja danych LOB, 469

konfigurowanie połączenia, 28

konkatenacja, 49

konstruktor, 352, 379

konto użytkownika, 32

konwersja

 kolekcji, 399
 napisu, 150
 niejawna, 464
 tabeli, 400
 typów, 128
 wysokość DataGodzina, 143

kończenie

 połączenia, 88
 transakcji, 252

kopiowanie

 danych, 452, 455–459
 wierszy, 245

koszt wykonania zapytań, 480

kursory, 322, 325

kursory bez ograniczenia, 327

kwalifikowane odwołania, 473

L

liść, 182
 LOB, 425
 BFILE, 426
 BLOB, 426
 CLOB, 426
 NCLOB, 426
 logiczna jednostka pracy, 251, 254
 logika warunkowa, 319
 lokalizator LOB, 426, 448

Ł

łączenie
 funkcji, 98
 operatorów zestawu, 175
 wartości, 49
 wywołań funkcji, 134
 z bazą danych, 40

M

manipulowanie
 tabelą zagnieżdżoną, 402
 tablicą VARRAY, 400
 metaznaki, 113, 114
 metoda
 APPEND(), 433
 CLOSE(), 433
 COMPARE(), 434
 COPY(), 435
 COUNT(), 404
 CREATETEMPORARY(), 435
 DELETE(), 406
 ERASE(), 436
 EXISTS(), 406
 EXTEND(), 407
 FILECLOSE(), 436
 FILECLOSEALL(), 437
 FILEEXISTS(), 437
 FILEGETNAME(), 437
 FILEISOPEN(), 438
 FILEOPEN(), 438
 FIRST(), 408
 FREETEMPORARY(), 439
 GET_STORAGE_LIMIT(), 440

GETCHUNKSIZE(), 439
 GETLENGTH(), 439
 INSTR(), 440
 ISOPEN(), 441
 ISTEMPORARY(), 441
 LAST(), 408
 LOADBLOBFROMFILE(), 443
 LOADFROMFILE(), 442
 NEXT(), 409
 OPEN(), 444
 PRIOR(), 410
 READ(), 445
 SUBSTR(), 445
 TRIM(), 411, 446
 WRITE(), 447
 WRITEAPPEND(), 447

metody
 mapujące, 397
 operujące na kolekcjach, 403
 pakietu DBMS_LOB, 431–433
 miejsce na tabeli, 30
 modyfikowanie
 danych, 429
 elementów
 kolekcji, 395
 tabeli zagnieżdżonej, 396
 tablicy VARRAY, 396
 indeksu, 305
 istniejących komórek, 229
 kolumny, 285
 perspektywy, 313
 sekwencji, 301
 wierszy, 38, 245

N

nagłówek, 85
 napisy
 formatujące datę, 132
 formatujące godzinę, 132
 narzędzia optymalizujące, 487
 narzędzie
 Automatic Database Diagnostic Monitor, 488
 Database Replay, 488
 Oracle Enterprise Manager, 487
 Real-Time SQL Monitoring, 488

SQL Access Advisor, 488
 SQL Performance Analyzer, 488
 SQL Plan Management, 489
 SQL Tuning Advisor, 488
 następstwo operatorów, 56
 nazwy stref czasowych, 143
 niejawna konwersja, 464
 nierównozłączenia, 61
 numery wierszy, 45

O

obcinanie tabeli, 292
 obiekty
 bazy danych, 349
 BFILE, 430
 BLOB, 428
 CLOB, 428
 dołączanie danych, 450
 kolumnowe, 352
 kopianie danych, 452
 NOT SUBSTITUTABLE, 371
 odczyt danych, 449
 tymczasowe, 452
 usuwanie danych, 453
 wyszukiwanie danych, 454
 obliczanie
 sum pośrednich, 86
 sumy kumulacyjnej, 213
 średniej centralnej, 215
 średniej kroczej, 214
 wektora bitowego, 198
 obliczenia na datach, 46
 obserwowanie operacji
 ALL STATEMENTS, 279
 BY ACCESS, 278
 BY SESSION, 278
 IN SESSION CURRENT, 279
 uprawnienia, 277
 USER_AUDIT_OBJECT, 279
 USER_AUDIT_SESSION, 279
 USER_AUDIT_STATEMENT, 279
 USER_AUDIT_TRAIL, 279
 WHENEVER NOT SUCCESSFUL, 278
 WHENEVER SUCCESSFUL, 278

- obsługa
 tabel zagnieżdżonych, 417
 wartości brakujących, 227
 wartości NULL, 227
- odbieranie
 roli, 276
 uprawnień, 271
 uprawnień roli, 276
 uprawnień systemowych, 265
- odczytywanie
 danych z CLOB, 449
 plików, 73
- odnajdywanie
 formacji, 240, 241
 wzorców, 237
- odpytywanie
 perspektyw, 308
 tabeli planu, 483
- odwołania
 do kolumn, 473
 obiektywne, 357
- odwrotne funkcje percentylu, 205
- ograniczenia złączeń
 zewnętrznych, 65
- okno SQL Developer, 28
- określanie formatu
 czasu, 133
 daty, 133
- operacje w planie wykonania, 485
- operand, 45
- operator
 \leq , 52
 \neq , 51
 $>$, 52
 ALL, 52, 163
 AND, 55
 ANY, 52, 162
 BETWEEN, 55
 CUBE, 209
 DISTINCT, 477
 EXISTS, 164, 477
 GROUPING SETS, 209
 IN, 54, 161, 418, 477
 INTERSECT, 174
 IS A SET, 422
 IS EMPTY, 422
 LIKE, 53
 MINUS, 174
 MULTISET, 419
- NOT EXISTS, 164, 165
 NOT IN, 165, 418
 OR, 56
 ROLLUP, 209
 SUBMULTISET, 419
 UNION, 173
 UNION ALL, 172
- operatorzy
 ANSI, 399
 arytmetyczne, 46
 do porównywania, 51
 jednowierszowe, 158
 logiczne, 55
 nierówności, 417
 równości, 417
 SQL, 53
 zestawu, 171
- optymalizacja, 480
- optymalizacja SQL, 471
- optymalizator, 486
- Oracle Enterprise Manager, 487
- Oracle PL/SQL, 41
- P**
- pakiet DBMS_LOB, 431
- pakiety, 337
- parametry formatujące
 daty i godziny, 129–131
 liczby, 110
- perspektywa user_varrays, 390
- perspektywy, 306
 zapisu obserwacji, 279
 złożone, 311
- pętla
 FOR, 227, 321, 325
 WHILE, 321
- PL/SQL, Procedural
 Language/SQL, 317
 bloki, 317
 duże obiekty, 431
 funkcje, 335
 instrukcja OPEN-FOR, 325
 kod maszynowy, 347
 kuratory, 322, 325
 logika warunkowa, 319
 metody, 403
 obiekty, 361
 obsługa błędów, 328
- piętle, 320
 predefiniowane wyjątki, 328
- procedury, 331
- rozszerzenia, 345
- sekwencje, 346
- typy, 319
- użycie kolekcji, 400
- zmienne, 319
- plan wykonania, 480, 483
 dla złączeń tabel, 484
- plik
 binaryContent.doc, 425, 426
 textContent.txt, 426
- pliki XML, 506
- pobieranie
 bieżącej daty, 142
 danych z CLOB, 428
 dat, 127
 elementów, 392, 393
 informacji, 43
 lokalizatora LOB, 448
 n-tego wiersza, 217
 wierszy, 322
 wszystkich kolumn, 44
- podstawianie nazw, 80
- podtyp, 369
- podzapytania
 jednowierszowe, 157
 rekurencyjne, 187
 skorelowane, 157, 163, 164
 w klauzuli FROM, 160
 w klauzuli HAVING, 159
 w klauzuli WHERE, 157
 wielokolumnowe, 157, 163
 wielowierszowe, 157, 161
 zagnieździone, 157, 166
- polecenie
 ACCEPT, 82
 ALTER SESSION, 134
 APPEND, 72
 BREAK ON, 86
 BTITLE, 86
 CHANGE, 72
 CLEAR, 76
 CLEAR] BUFFER, 72
 COLUMN, 77
 COMPUTE, 86
 DEFINE, 81
 DEL, 72

polecenie
 DESCRIBE, 351
 FORMAT, 76
 GET, 73
 HEADING, 76
 HELP, 87
 JUSTIFY, 76
 LIST, 72
 RUN, 72
 SAVE, 73
 SET DEFINE, 80
 SET DESCRIBE DEPTH, 352
 SET VERIFY, 80
 SPOOL, 73
 START, 73
 TTITLE, 86
 WORD_WWRAPPED, 76
 WRAPPED, 76
połączenie z bazą danych, 28, 88
pomoc SQL*Plus, 87
porównywanie
 planów wykonania, 485
 wartości, 51
 wartości obiektów, 359
poziomy izolacji, 256
predykat IS ANY, 225
procedura, 331
 delete_product(), 365
 display_product(), 362
 insert_product(), 363
 nclob_example(), 464
 product_lifecycle(), 366
 update_product(), 364
 update_product_price(), 363
procedury PL/SQL, 448
program SQL*Plus, 25, 71
programowanie, 317
przeciążanie metod, 381
przeglądanie
 planów wykonania, 481
 struktury tabeli, 71
przekształcanie kolumn, 463
przesłanianie metod, 382
przestawianie, 231
przestrzeń tabel, 261, 416
przesunięcie strefy czasowej, 142
przesyłanie wskazówek, 486
przygotowywanie podzapytań,
 168

przyznawanie
 roli, 272
 uprawnień, 257, 266
 uprawnień roli, 272
pseudokolumna LEVEL, 183
punkty zachowania, 252

R

raport, 83
raportowanie sumy, 218
Real-Time SQL Monitoring, 488
rekurencyjne podzapytania
 przygotowywane, 187
relacja nadzędny/podrzędny, 36
relacyjna baza danych, 23
rodzaje
 dużych obiektów, 426
 podzapytań, 157
role, 261, 271
 aktywacja, 276
 deaktywacja, 276
 odbieranie uprawnień, 276
 przyznawanie uprawnień, 272
sprawdzanie, 272
 uprawnienia obiektowe, 274
 uprawnienia systemowe, 273
usuwanie, 277
 zastosowanie uprawnień, 275
rozłączanie z bazą danych, 40
rozmiar
 strony, 77
 typu elementu, 415
 wiersza, 78
rozpoczynanie transakcji, 252
rozszerzenia
 kolekcji, 414
 PL/SQL, 345
równozłączenia, 61

S

scalanie wierszy, 249
schemat, 23, 30, 33
schemat XML, 506, 508, 511
sekwencje, 296
składowanie
 dat, 127
 wierszy, 480

skrypt, 30, 31
 collection_schema.sql, 387
 collection_schema2.sql, 412
 collection_schema3.sql, 414
 object_schema.sql, 350
 object_schema2.sql, 368
 object_schema3.sql, 384
 store_schema.sql, 34–37

słów kluczowych

 AND, 225
 BETWEEN, 225
 PRIMARY KEY, 34
 USING, 67
specyfikacja pakietu, 338
sprawdzanie
 ról przyznanych, 272
 uprawnień, 264–268, 273
SQL, Structured Query Language,
 24
SQL Access Advisor, 488
SQL Developer, 27
SQL Performance Analyzer, 488
SQL Plan Management, 489
SQL Tuning Advisor, 488
SQL*Plus, 25, 71
SQL/92, 66
standard ANSI, 59, 417
statystyki tabeli, 485
stopka, 85
strefa czasowa, 140
 bazy danych, 141
 sesji, 141, 142
struktura tabeli, 71
strukturny język zapytań, 24
suma kumulacyjna, 213
sumy pośrednie, 86
synonimy, 270
synonimy publiczne, 270
system zarządzania, 24
szyfrowanie danych
 kolumny, 468
 LOB, 465, 466

Ś

średnia
 centralna, 215
 krocząca, 214
średnik, 71

T

tabela, 23, 281
 customers, 29
 planu, 481, 483
 tabele
 dodawanie komentarza, 291
 kolumny, 285
 kolumny wirtualne, 284
 nadrzędne, 36, 247
 obcinanie, 292
 obiektywne, 354
 pobieranie informacji, 282
 podrzędne, 35, 247
 tymczasowe, 416
 usuwanie, 292
 więzy, 286
 więzy odroczone, 289
 zagnieżdżone, 387–390, 402
 zmienianie, 284
 zmienianie nazwy, 291
 tablica VARRAY, 389, 393, 400
 liczba elementów, 416
 manipulowanie, 400
 tablice asocjacyjne, 387, 415
 TC, Transaction Control, 24
 transakcja SERIALIZABLE, 256
 transakcje bazodanowe
 ACID, 254
 blokowanie, 255
 kończenie, 252
 poziomy izolacji, 256
 punkty zachowania, 252
 rozpoczynanie, 252
 współbieżne, 255
 wycofywanie, 251
 zatwierdzanie, 251
 tworzenie
 archiwów migawek, 314
 funkcji, 336
 indeksu, 281
 bitmapowego, 305
 opartego na funkcji, 303
 typu B-drzewo, 303
 kolekcji, 388
 konta użytkownika, 32, 262
 obiektu katalogu, 430
 perspektyw, 281, 307, 309
 perspektyw złożonych, 311
 portfela, 465

procedury, 332
 raportów, 83
 ról, 271
 schematu, 30
 schematu bazy danych, 368
 schematu XML, 506
 sekwencji, 281, 296
 specyfikacji pakietu, 338
 synonimów, 270
 tabeli, 281
 tabeli planu, 482
 tabeli zagnieżdżonej, 388
 treści pakietu, 338
 typów obiektywnych, 350
 typu VARRAY, 388
 użytkownika, 30
 wyzwalacza, 341
 typ danych
 BFILE, 516
 BINARY_DOUBLE, 292, 515
 BINARY_FLOAT, 292, 515
 BINARY_INTEGER, 517
 BLOB, 516
 BOOLEAN, 517
 CHAR, 33, 515
 CLOB, 516
 DATE, 33, 516
 DEC, 516
 DECIMAL, 516
 DOUBLE PRECISION, 516
 FLOAT, 516
 INT, 516
 INTEGER, 33, 516
 INTERVAL DAY, 153, 516
 INTERVAL YEAR, 152, 516
 LONG, 462, 517
 LONG RAW, 426, 462, 517
 NATURAL, 517
 NATURALN, 517
 NCHAR, 515
 NCLOB, 516
 NUMBER, 33, 515, 516
 NUMERIC, 515, 516
 NVARCHAR2, 515
 PLS_INTEGER, 517
 POSITIVE, 517
 POSITIVEN, 517
 RAW, 517
 REAL, 516
 RECORD, 518
 REF, 517
 REF CURSOR, 518
 ROWID, 517
 SIGNTYPE, 517
 SIMPLE_INTEGER, 345, 517
 SMALLINT, 516
 STRING, 518
 TIMESTAMP, 144, 516
 TIMESTAMP WITH LOCAL
 TIME ZONE, 145, 150
 TIMESTAMP WITH TIME
 ZONE, 145
 TO MONTH, 516
 TO SECOND, 516
 UROWID, 517
 VARCHAR2, 33, 515
 VARRAY, 387, 517
 XMLType, 517
 typy
 datowników, 144
 funkcji, 91
 interwałów czasowych, 151
 LOB, 425
 obiektywne NOT
 INSTANTIABLE, 378
 nadrużne, 369
 złączeń, 61
 uruchamianie
 plików, 73
 skryptu, 31
 SQL*Plus, 25, 26
 wyzwalacza, 343
 ustanowienie połączenia, 30

U

ustawianie rozmiaru

strony, 77

wiersza, 78

usuwanie

danych z obiektu, 453

funkcji, 337

indeksu, 305

konta użytkownika, 263

pakietu, 340

perspektywy, 313

procedury, 335

rolи, 277

sekwencji, 302

tabeli, 292

użytkownika, 30

wierszy, 38, 40, 246

więzów, 288

wyzwalaacza, 345

użycie

aliasów kolumn, 48

aliasów tabel, 59

atrybutu

new, 464

dołączanych baz danych, 31

dużych obiektów, 428, 431

funkcji

agregujących, 121, 123

CAST(), 399, 400

CUME_DIST(), 211

CURRENTV(), 226

DECODE(), 177

FIRST(), 221

GROUP_ID(), 200

GROUPING(), 195, 197

GROUPING_ID(), 198

hipotetycznego rankingu
i rozkładu, 222

LAG(), 220

LAST(), 221

LEAD(), 220

LISTAGG(), 220

NTILE(), 211

okna, 212

PERCENT_RANK(), 211

RANK(), 206

DENSE_RANK(), 206

RATIO_TO_REPORT(),

219

regresji liniowej, 221

ROW_NUMBER(), 211

TABLE(), 394, 395

TRANSLATE(), 176

IGNORE NAV, 229

KEEP NAV, 229

IS PRESENT, 228

klaузuli

DEFAULT ON NULL, 293

FETCH FIRST, 234

MODEL, 223

OFFSET, 235

PARTITION BY, 208

PERCENT, 236

PIVOT, 230

ROLLUP, 192

UNPIVOT, 230, 233

WITH TIES, 236

kolekcji, 389, 400

metody mapującej, 397

obiektów, 361

BFILE, 430

BLOB, 428

CLOB, 428

odwrotnych funkcji

rankingowych, 212

operatora

ALL, 163

ANY, 162

CUBE, 209

EXISTS, 164

GROUPING SETS, 209

IN, 161

NOT EXISTS, 164, 165

ROLLUP, 209

perspektyw, 307

perspektyw złożonych, 311

podtypu, 370

pseudokolumny LEVEL, 183

SCN, 259

sekwencji, 298, 300

SQL*Plus, 25

tymczasowych obiektów, 452

typów obiektowych, 352

typu

BINARY_DOUBLE, 293

BINARY_FLOAT, 293

TIMESTAMP, 144, 145

VARRAY, 389

wartości domyślnych, 248

wyrażeń CASE, 178, 196, 473

wyrażeń z funkcjami, 98

zapytań retrospektywnych,

257

złączeń tabel, 472

zmiennych, 79

dowiązanych, 480

tymczasowych, 83

zdefiniowanych, 84

użytkownicy, 261

hasło, 263

konto, 262

odbieranie uprawnień

obiektowych, 271

role, 272

sprawdzanie uprawnień, 264

uprawnienia obiektowe, 266

uprawnienia systemowe, 263

usuwanie konta, 263

W

wartości domyślne, 248

wartości obiektów, 359

wartość NULL, 34, 49, 208, 227

wartość wymiaru, 226

węzeł

główny, 181

nadrzędny, 181

podrzędny, 182

węzły równorzędne, 182

widok wbudowany LATERAL,

203

wiersze, 38

więzy, 286

CHECK, 286

CHECK OPTION, 309

FOREIGN KEY, 287

NOT NULL, 287

odroczone, 289

READ ONLY, 310

UNIQUE, 288

właściwości

dużych obiektów, 463, 465,

469

transakcji, 254

włączanie

więzów, 288

wyzwalaacza, 345

wskaazywanie wierszy, 44
 wskaźnik do pliku, 430
 współbieżne transakcje, 255
 wstawianie
 danych, 462
 wierszy, 243
 wycofywanie transakcji, 251
 wyjątek, 328
 DUP_VAL_ON_INDEX, 330
 INVALID_NUMBER, 330
 OTHERS, 331
 ZERO_DIVIDE, 330
 wyjątki predefiniowane, 328
 wykonywanie złączeń, 66
 krzyżowych, 70
 wewnętrznych, 66–68
 własnych, 69
 zewnętrznych, 68
 lewostronnych, 68
 pełnych, 69
 prawostronnych, 69
 wyłączanie więzów, 288
 wymuszanie więzów
 klucza głównego, 247
 kluczy obcych, 247
 wypisywanie zmiennych, 81
 wyrażenia
 CASE, 178, 196, 473
 regularne, 113, 115
 wyszukiwanie danych, 454
 wyświetlanie unikatowych
 wierszy, 50
 wywoływanie
 funkcji, 336, 339
 procedur, 333, 339
 wyzwalacze, 340
 tworzenie, 341
 uruchamianie, 343
 usuwanie, 345
 włączanie, 345
 wyłączanie, 345
 wzorce, 237

X

XML, Extensible Markup
 Language, 491

Z

zapis
 mieszany, 334
 nazywany, 334
 pozycyjny, 224
 symboliczny, 224
 zapisywanie
 danych XML, 501, 506
 do obiektu CLOB, 450
 plików, 73
 zapytania, 24
 hierarchiczne, 181, 183, 187
 jednowierszowe, 160
 o liczbę wierszy, 234
 retrospektywne
 uprawnienia, 257
 użycie SCN, 259
 w oparciu o czas, 258
 zaawansowane, 171
 zastosowanie uprawnień
 obiektowych, 269
 rolи, 275
 systemowych, 265
 zatwierdzanie transakcji, 251
 zbieranie statystyk tabeli, 485
 zestaw wyników, 44
 złączenia, 61, 66
 krzyżowe, 70
 wewnętrzne, 61, 66–68
 własne, 61, 65, 69
 zewnętrzne, 61, 68
 lewostronne, 63
 prawostonne, 63
 złączenie tabel, 58
 złożony klucz główny, 37
 zmienianie
 edytora, 75
 hasła, 263
 nazwy tabeli, 291
 pozycji kolumn, 193
 rozmiaru typu elementu, 415
 tabeli, 284
 zawartości tabeli, 243
 znaku, 80
 zmienna środowiskowa
 NLS_LANG, 27
 zmienne, 79
 dowiązane, 478
 tymczasowe, 79
 zdefiniowane, 81
 znacznik czasu, 127, 143