

Oracle SQL Developer

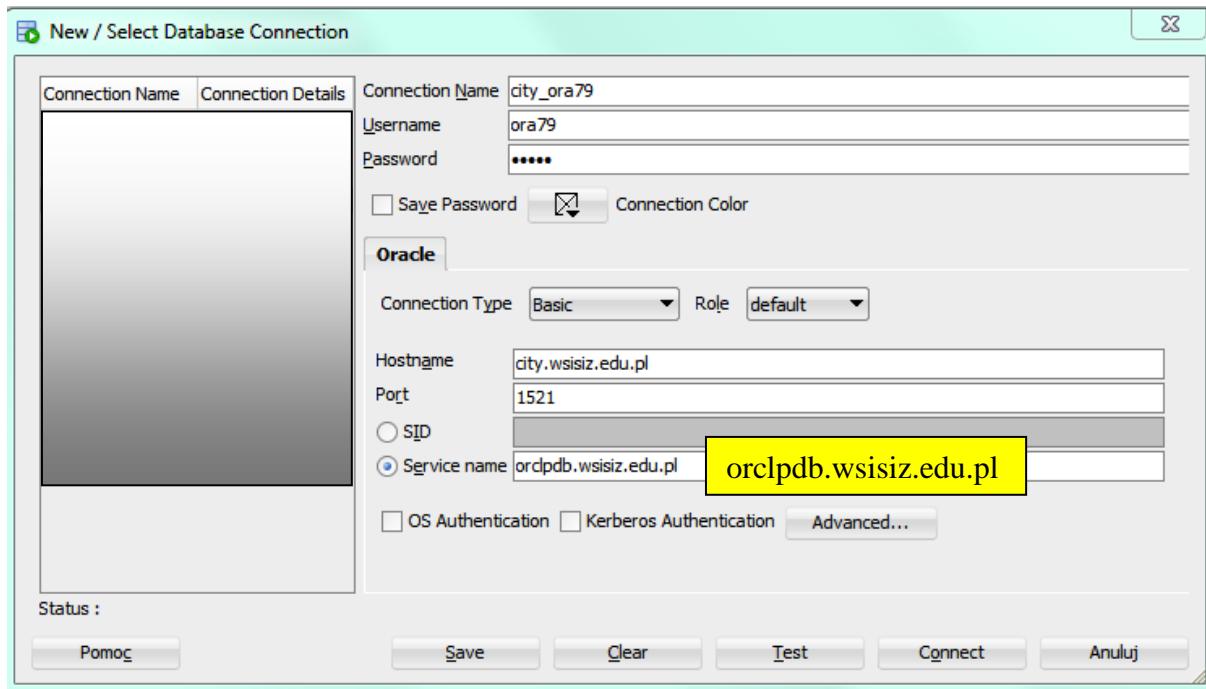
Jest to narzędzie umożliwiające łączenie się z bazami danych na serwerach Oracle, zarządzanie tymi bazami przy pomocy języka SQL oraz programowanie serwerów przy pomocy języka PL/SQL.

Oprogramowanie można pobrać ze strony producenta:

<https://www.oracle.com/tools/downloads/sqldev-downloads.html>

wybierając odpowiednią wersję systemu operacyjnego.

W celu logowania się do bazy danych na określonym serwerze należy zdefiniować połaczenie (Connect), tak jak na poniższym rysunku:

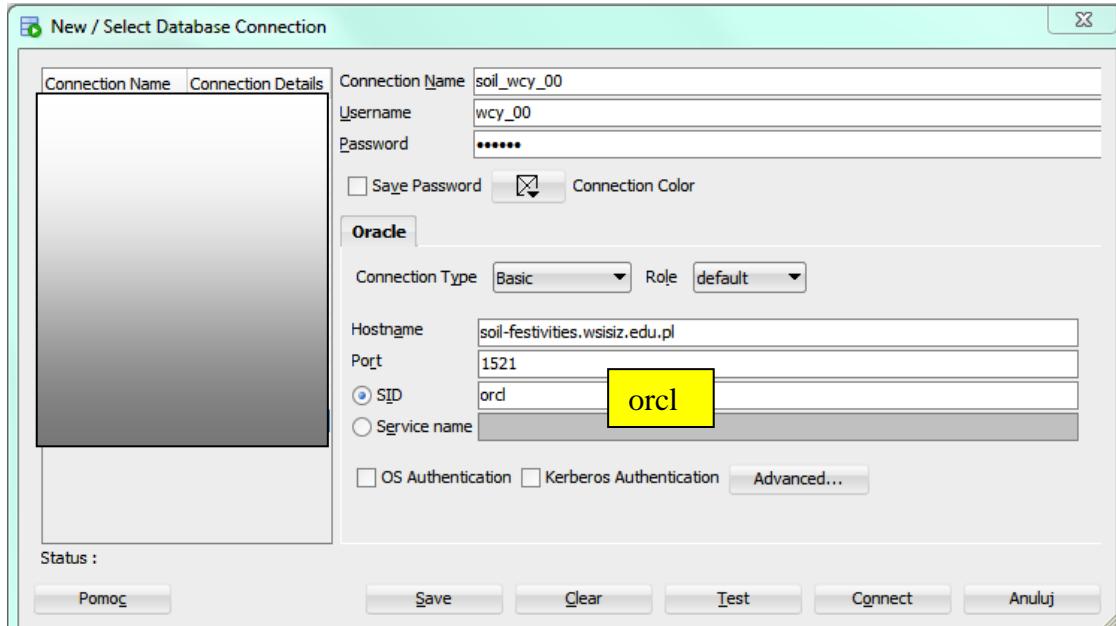


gdzie:

- Connection Name - własna nazwa połączenia,
- Username - nazwa konta (schematu) nadana przez administratora,
- Password - hasło dostępu do konta (schematu),
- Hostname - nazwa komputera, na którym znajduje się serwer Oracle,
- Port - standardowy port Oracle ma wartość 1521,
- SID - nazwa bazy danych Oracle na wskazanym serwerze,
- Service name - nazwa serwisu, przy pomocy którego następuje połączenie z bazą danych.

Drugim sposobem łączenia się z bazą Oracle jest wskazanie nazwy bazy jako *SID* zamiast nazwy serwisu *Service name*.

Przykładowo chcąc zalogować się na inny serwer Oracle ekran definicji połączenia może wyglądać tak:



gdzie zamiast nazwy serwisu należy podać wygenerowaną w procesie instalacji nazwę bazy danych, w tym przypadku *orcl*.

SQL Plus

Dodatkowym narzędziem używanym do zarządzania bazą Oracle jest SQL Plus. Szczególnie często jest wykorzystywane do administrowania serwerami Oracle, do wdrażania gotowych rozwiązań oraz do testowania oprogramowania PL/SQL i SQL.

Wymaga zainstalowania oprogramowania Client Oracle na odpowiedni system operacyjny (na przykład <https://www.oracle.com/technetwork/topics/winx64soft-089540.html>) oraz zdefiniowania pliku konfiguracyjnego *tnsnames.ora* zawierającego definicje połączeń z bazami i serwerami Oracle.

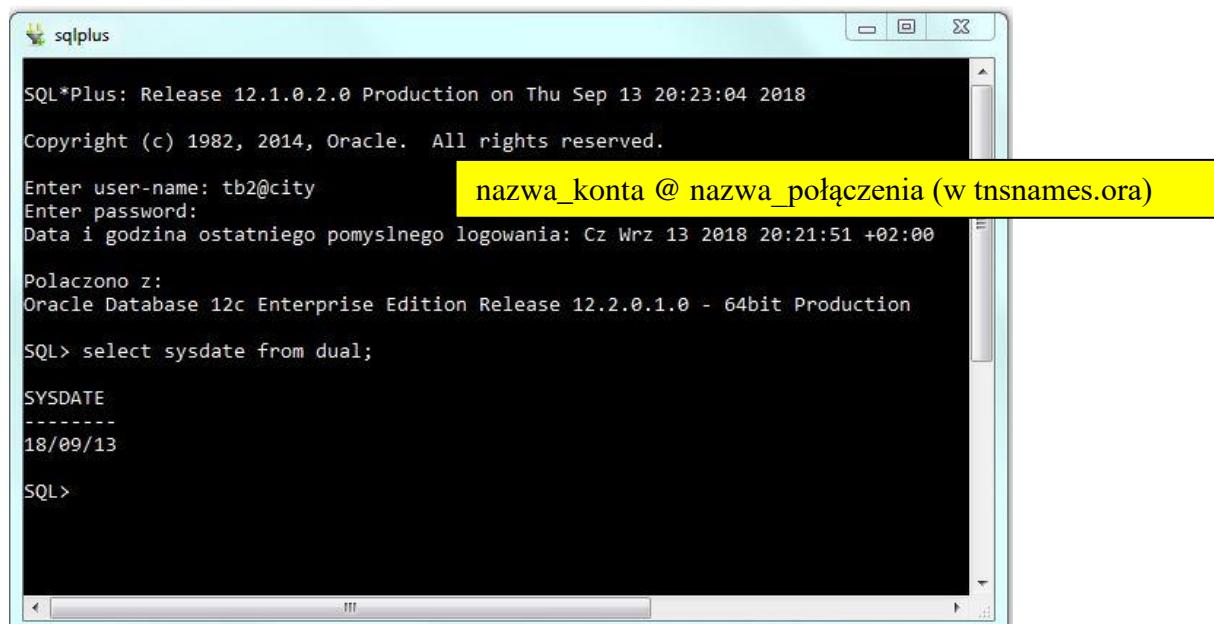
Przykładowy plik konfiguracyjny może wyglądać tak:

```

soil =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = soil-festivities.wsisiz.edu.pl)(PORT = 1521))
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = orcl)
  )
)
city =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = city.wsisiz.edu.pl)(PORT = 1521))
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = orclpdb.wsisiz.edu.pl)
  )
)

```

Fragment sesji realizowanej przy użyciu SQL Plus:



SQL*Plus: Release 12.1.0.2.0 Production on Thu Sep 13 20:23:04 2018
Copyright (c) 1982, 2014, Oracle. All rights reserved.
Enter user-name: tb2@city nazwa_konta @ nazwa_łączenia (w tnsnames.ora)
Enter password:
Data i godzina ostatniego pomyslnego logowania: Cz Wrz 13 2018 20:21:51 +02:00
Polaczono z:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production
SQL> select sysdate from dual;
SYSDATE

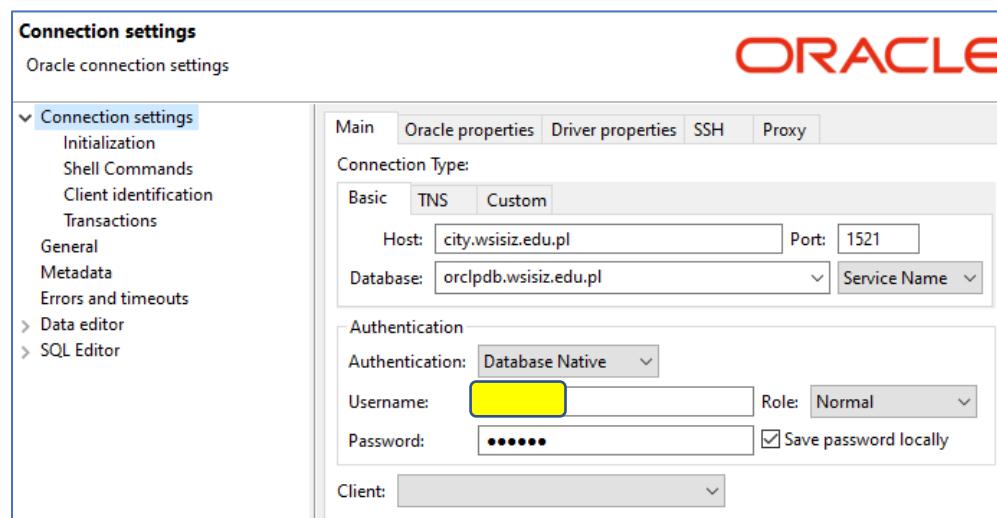
18/09/13
SQL>

DBeaver – Universal Database Manager



Jest to interesujące (darmowe w szerokim zakresie) narzędzie służące do zarządzania różnymi systemami baz danych, nie tylko Oracle. Definiowanie połączeń z serwerami bazodanowymi odbywa się w oparciu o drivery jdbc, które są wbudowane w aplikację.

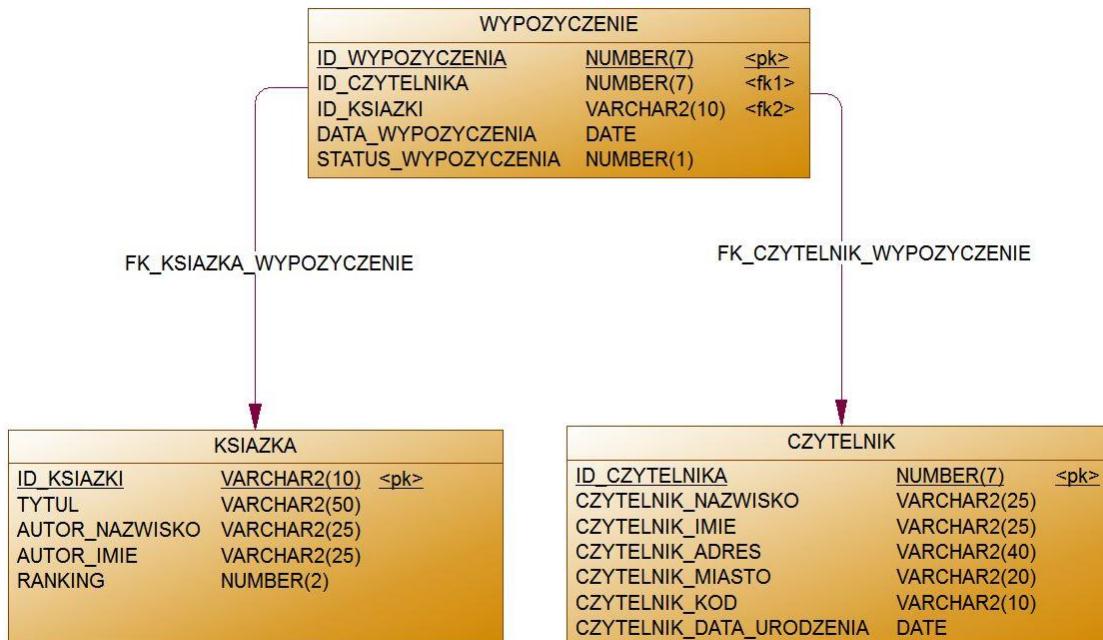
Szczegóły oraz oprogramowanie można znaleźć na stronie <https://dbeaver.io/>



The screenshot shows the 'Connection settings' dialog for an Oracle connection. The main pane displays the 'Main' tab of the configuration. The 'Connection Type' section is set to 'TNS'. The 'Host' field contains 'city.wsisiz.edu.pl' and the 'Port' field contains '1521'. The 'Database' field contains 'orclpdb.wsisiz.edu.pl'. The 'Authentication' section shows 'Database Native' selected for authentication, with 'Username' and 'Password' fields filled in. A checkbox for 'Save password locally' is checked. On the left sidebar, there are sections for 'Connection settings' (Initialization, Shell Commands, Client identification, Transactions, General, Metadata, Errors and timeouts), 'Data editor', and 'SQL Editor'. The 'ORACLE' logo is visible in the top right corner of the dialog.

Relacyjny model bazy danych – koncepcja

Poniżej został przedstawiony model relacyjny pewnego fragmentu rzeczywistości związanego z działalnością biblioteki opracowany przy pomocy SAP PowerDesigner.¹



Mamy do czynienia z trzema tabelami, z których dwie **KSIĄZKA** i **CZYTELNIK** mają charakter statyczny obrazujący ewidencję książek i czytelników w bibliotece. Trzecia tabela **WYPOŻYCZENIE** rejestruje fakty związane z procesem wypożyczenia, a więc ma charakter dynamiczny.

Każda tabela posiada ścisłe określona strukturę, na którą składa się przede wszystkim jej nazwa oraz specyfikacja kolumn. Przykładowo tabela **KSIĄZKA** zawiera pięć kolumn o określonych nazwach i typach. Nazwy kolumn muszą być unikalne w ramach definicji tabeli czyli mogą się powtarzać w innych tabelach. Typy kolumn określają rodzaj danych w nich przechowywanych i w wielu przypadkach zależą od konkretnego systemu zarządzania bazą danych, w którym się je implementuje. Nie mniej jednak można wskazać kilka podstawowych typów uniwersalnych, takich jak: *varchar* (lub *char*) określający typy znakowe (np. nazwiska i tytuły)², *number* określający typ liczbowy (np. kwoty lub miliary) oraz *date* związany z zapamiętywaniem dat.

Typy znakowe muszą mieć zdefiniowaną skalę czyli swoją długość, np. *varchar(10)* oznacza, że w kolumnie takiego typu można umieścićłańcuch znaków o maksymalnej długości 10 znaków.

Typy liczbowe także mogą mieć zdefiniowaną skalę, np. *number(2)* oznacza możliwość przechowywania liczb całkowitych z zakresu [-99..+99]. Dodatkowo typy liczbowe mogą zawierać precyzję, np. *number(5,2)* oznacza możliwość przechowywania liczb z dokładnością do dwóch miejsc po przecinku [-999.99..+999.99]. Ale również mogą występować typy liczbowe bez skali i precyzji. Należą do nich na przykład *integer* i *float*, chociaż nie jest to zalecane w przypadku definiowania tabel. Typ *date* umożliwia przechowywanie danych interpretowanych jako daty. Należy pamiętać, że format przechowywanej daty musi być zgodny z formatem daty zdefiniowanym na serwerze bazodanowym, np. jeśli na serwerze data systemowa jest zdefiniowana jako YY/MM/DD, to w takim samym formacie należy datę wprowadzać do tabeli lub używać specjalnych funkcji konwersji.

¹ PowerDesigner firmy SAP (poprzednio Sybase) jest narzędziem służącym do modelowania systemów czyli tworzenia modeli biznesowych, logicznych i fizycznych w różnych środowiskach (Oracle, Sybase, MS Server i wielu innych).

² Dla bazy danych Oracle obowiązującym typem znakowym jest *varchar2*.

Każda tabela posiada jedną (na ogólnie) kolumnę specjalnie wyróżnioną zwaną kluczem głównym.

Klucz główny (Primary Key) – jedna lub kilka kolumn w tabeli, na podstawie których system zarządzania bazą danych kontroluje dane podczas ich wprowadzania. W tabeli nie można zapisać dwóch wierszy mających takie same wartości w kolumnach klucza głównego. Przykładowo w tabeli CZYTELNIK nie można wprowadzić danych dwóch czytelników mających tę samą wartość w kolumnie ID_CZYTELNIKA. Można wprowadzić dwa takie same nazwiska, lecz różniące się ID_CZYTELNIKA. **Klucz główny zapewnia, że w tabeli nie ma dwóch takich samych wierszy (zapewnia niepowtarzalność danych w tabeli).**

Klucz główny może być kluczem złożonym składającym się z kilku kolumn tabeli. W takim przypadku kombinacja wartości kolumn wchodzących w skład klucza głównego musi być niepowtarzalna.

Na powyższym diagramie kolumny będące kluczami głównymi oznaczone są atrybutem <pk>. Brak jest klucza złożonego, ale przy niewielkiej modyfikacji modelu można taki klucz zdefiniować. W tabeli WYPOZYCZENIE zamiast kolumny ID_WYPOZYCZENIA będącej kluczem głównym można zastosować złożony klucz główny składający się z trzech kolumn: ID_CZYTELNIKA, ID_KSIAZKI oraz DATA_WYPOZYCZENIA, a kolumnę ID_WYPOZYCZENIA usunąć. Oznaczałoby to, że nie jest możliwe, aby ten sam czytelnik tę samą książkę mógł wypożyczyć więcej niż raz tego samego dnia.

Kolejnym ważnym pojęciem związanym z relacyjnym modelem jest klucz obcy.

Klucz obcy (Foreign Key) – definiuje relację między dwiema tabelami zapewniając integralność danych. Przykładowo w tabeli KSIAZKA są trzy wiersze: (1, W pustyni i w puszczy,...), (2, Ogniem i mieczem,) i (3, Potop....). Rejestrując fakt wypożyczenia książki przez zarejestrowanego czytelnika czyli wprowadzając wiersz do tabeli WYPOZYCZENIE nie można podać w tym wierszu na pozycji ID_KSIAZKI wartości 4, gdyż książki o takim identyfikatorze nie ma w tabeli KSIAZKA. Najpierw należy do tabeli KSIAZKA wprowadzić tę pozycję (4, Pan Wołodyjowski....), a następnie ją wypożyczyć wprowadzając stosowny wiersz do tabeli WYPOZYCZENIE. Mówimy, że dane w bazie danych są zintegrowane czyli spójne. Zapewnia to klucz obcy.

Podobnie nie można wypożyczyć żadnej książki czytelnikowi, który nie jest zarejestrowany.

Na powyższym diagramie klucze obce zaznaczone są jako:

- <fk1> - klucz obcy między tabelami WYPOZYCZENIE i CZYTELNIK
(relacja FK_CZYTELNIK_WYPOZYCZENIE),
- <fk2> - klucz obcy między tabelami WYPOZYCZENIE i KSIAZKA
(relacja FK_KSIAZKA_WYPOZYCZENIE).

Należy zwrócić uwagę, że dwie kolumny, poprzez które tabele są w relacji do siebie mają te same nazwy (ale mogą być różne) i te same typy. Przykładem jest kolumna ID_CZYTELNIKA, która występuje w tabelach CZYTELNIK (jako klucz główny) i WYPOZYCZENIE i w obu przypadkach jest tego samego typu.

DDL³ – zdania SQL tworzące i modyfikujące strukturę bazy danych

Podstawowymi zdaniami z tej grupy zdań SQL są zdania *create*, *alter*, *rename* oraz *drop*.

Zdanie *create* służy do tworzenia wszelkich obiektów bazodanowych, a w szczególności tabel. Ogólna postać zdania:

```
create table nazwa_tabeli ( kol_a          typ_danych Primary Key,
                            kol_b          typ_danych,
                            .....          .....,
                            kol_n          typ_danych,
                            Foreign Key (kol_b) References nazwa_innej_tabeli (kol_c_innej_tabeli)
                          );
```

³ DDL – Data Definition Language

Sposoby definiowania klucza głównego:

1. Na poziomie kolumny:

.....
kol_a typ_danych *Primary Key*,
.....

2. Na poziomie tabeli:

.....
kol_a typ_danych,
.....
kol_ostatnia typ_danych,
Primary Key(kol_a),
.....

3. Klucz złożony (tylko na poziomie tabeli):

.....
kol_a typ_danych,
kol_b typ_danych,
.....
kol_ostatnia typ_danych,
Primary Key(kol_a, kol_b),
.....

Definiowanie klucza obcego:

Jest kilka sposobów definiowania klucza obcego:

1. Zapis na końcu definicji struktury tabeli:

```
create table nazwa_tabeli ( kol_a typ_danych Primary Key,
                            kol_b typ_danych,
                            .....  
kol_n typ_danych,  
Foreign Key (kol_b) References nazwa_innej_tabeli (kol_c_innej_tabeli)  
);
```

przy czym:

- w momencie jego definiowania tabela *nazwa_innej_tabeli* musi istnieć w schemacie,
- *kol_c_innej_tabeli* jest kluczem głównym w tabeli *nazwa_innej_tabeli*.

2. Definicja na poziomie kolumny:

```
create table nazwa_tabeli (.....,  
                            kol_b typ_danych References nazwa_innej_tabeli (kol_b_innej_tabeli),  
                            .....  
                            kol_n typ_danych  
);
```

3. Przy użyciu zdania *alter* (przykład poniżej).

Język SQL posiada odpowiednie konstrukcje zdaniowe (*alter*, *rename*, *drop*) do modyfikacji już zaprojektowanego i zaimplementowanego modelu danych. Modyfikacje te mogą polegać na usuwaniu, zmianie nazwy kolumn i tabel, dodawaniu kolumn, zmianie typów danych istniejących kolumn, jak również definiowaniu referencji czyli kluczy obcych. Poniżej przedstawione zostały na przykładach podstawowe zdania języka SQL umożliwiające modyfikacje modelu danych.

Usunięcie kolumny z tabeli:

```
alter table KSIAZKA  
    drop column RANKING;
```

Dodanie kolumny do tabeli:

```
alter table KSIAZKA  
    add RANKING varchar2(3);
```

Modyfikacja istniejącej kolumny w tabeli (typu danych):

```
alter table KSIAZKA  
    modify RANKING varchar2(5);
```

Zmiana nazwy kolumny w tabeli:

```
alter table KSIAZKA  
    rename column RANKING to OCENA;
```

Zmiana nazwy tabeli:

```
rename KSIAZKA to KSIAZKA_ZMIANA;
```

Zdefiniowanie referencji:

```
alter table WYPOZYCZENIE  
    add constraint FK_CZYTELNIK_WYPOZYCZENIE foreign key (ID_CZYTELNIKA)  
        references CZYTELNIK (ID_CZYTELNIKA);
```

przy czym zakłada się, że obie tabele (*WYPOZYCZENIE* i *CZYTELNIK*) istnieją w schemacie.

Usunięcie referencji:

```
alter table WYPOZYCZENIE  
    drop constraint FK_CZYTELNIK_WYPOZYCZENIE;
```

Usunięcie tabeli z bazy danych:

```
drop table KSIAZKA_ZMIANA;
```

DML⁴ – zdania SQL umożliwiające wprowadzanie, aktualizację i usuwanie danych z tabel

Podstawowymi zdaniami tej grupy są zdania: *insert*, *update* oraz *delete*.

Zdanie *insert* służy do wprowadzania wierszy do tabeli. Jest kilka sposobów użycia tego zdania:

1. Wprowadzanie danych w kolejności zgodnej ze strukturą danych:

```
insert into KSIAZKA  
    (ID_KSIAZKI, TYTUL, AUTOR_NAZWISKO, AUTOR_IMIE, RANKING)
```

⁴ DML – Data Manipulation Language

values

(15, 'Pan Wołodyjowski', 'Sienkiewicz', 'Henryk', 10);

2. Uproszczony wariant sposobu pierwszego:

insert into KSIAZKA

values

(15, 'Pan Wołodyjowski', 'Sienkiewicz', 'Henryk', 10);

3. Wprowadzanie danych w kolejności niezgodnej ze strukturą danych:

insert into KSIAZKA

(ID_KSIAZKI, AUTOR_NAZWISKO, AUTOR_IMIE, TYTUL, RANKING)

values

(15, 'Sienkiewicz', 'Henryk', 'Pan Wołodyjowski', 10);

4. Wprowadzanie niepełnych danych:

a).

insert into KSIAZKA

(ID_KSIAZKI, AUTOR_NAZWISKO, TYTUL)

values

(15, 'Sienkiewicz', 'Pan Wołodyjowski');

b).

insert into KSIAZKA

values

(15, 'Pan Wołodyjowski', 'Sienkiewicz', ' ', 10);

W tym ostatnim przypadku dane są wprowadzane nie do wszystkich kolumn tabeli (porównaj ze strukturą tabeli KSIAZKA). W wariantie 4a wyspecyfikowane są wybrane kolumny tabeli i odpowiadające im wartości, a w wariantie 4b wyspecyfikowane są domyślnie wszystkie kolumny, ale we frazie *values* brak danych reprezentowany jest przez pusty parametr (' ').

Zdanie *update* służy do modyfikacji pojedynczego wiersza lub grupy wierszy już istniejących w tabeli.

Przykładowo zdanie:

*update KSIAZKA
set RANKING = 9
where ID_KSIAZKI = 5;*

spowoduje, że w tabeli KSIAZKA egzemplarz biblioteczny o numerze 5 otrzyma ocenę rankingową 9. Ponieważ ID_KSIAZKI jest kluczem głównym tabeli KSIAZKA, więc jest pewność, że tylko jedna pozycja książkowa zostanie tym zdaniem zmodyfikowana.

Zdanie

*update KSIAZKA
set RANKING = 9
where AUTOR_NAZWISKO = 'Kraszewski';*

spowoduje, że wszystkie książki Kraszewskiego otrzymają ranking 9, gdyż w kolumnie AUTOR_NAZWISKO zapis 'Kraszewski' może powtarzać się wielokrotnie, nie jest ona bowiem kolumną klucza głównego. Natomiast zdanie:

*update KSIAZKA
set RANKING = 9;*

spowoduje, że wszystkie pozycje książkowe w tabeli KSIAZKA otrzymają ranking 9.

Zdanie

```
update KSIAZKA  
    set AUTOR_NAZWISKO ='Sienkiewicz',  
        AUTOR_IMIE = 'Henryk'  
    where TYTUL = 'W pustyni i w puszczy';
```

umożliwia modyfikacje danych w kilku kolumnach jednocześnie.

Zdanie *delete* służy do usuwania pojedynczego wiersza lub grupy wierszy już istniejących w tabeli.

Przykładowo zdanie:

```
delete WYPOZYCZENIA  
    where DATA_WYPOZYCZENIA < '2021/01/01';
```

spowoduje, że z tabeli WYPOZYCZENIA zostaną usunięte wszystkie informacje o wypożyczeniach dokonanych do końca 2020 roku.

Zdanie

```
delete WYPOZYCZENIA  
    where ID_WYPOZYCZENIA = 4;
```

spowoduje usunięcie jednego wiersza z tabeli, gdyż ID_WYPOZYCZENIA jest kluczem głównym.

Zdanie

```
delete WYPOZYCZENIA;
```

usunie z tabeli WYPOZYCZENIA wszystkie wiersze.

Z używaniem zdań języka DML (*insert*, *update* i *delete*) nierozerwalnie związane są dwa zdania: *commit* (zatwierdzenie transakcji) i *rollback* (cofnięcie transakcji).

Poniżej zostanie przedstawiony scenariusz użycia obu tych zdań:

1. W tabeli WYPOZYCZENIA znajduje się pewna liczba wierszy (np. 20),
2. Użycie zdania:

```
delete WYPOZYCZENIA;
```

spowoduje usunięcie z tabeli wszystkich wierszy,

3. Użycie zdania *rollback* cofa skutki działania wszystkich zdań DML od ostatniego *commit* lub od początku sesji, w tym przypadku powyższego zdania czyli nadal jest 20 wierszy,
4. Użycie zdania:

```
delete WYPOZYCZENIA  
    where DATA_WYPOZYCZENIA > '2021/03/01';
```

spowoduje usunięcie pewnej grupy wierszy (np. 5),

5. Użycie zdania *commit* zatwierdzi powyższe zdanie i w tabeli WYPOZYCZENIA na trwałe będzie 15 wierszy.
6. Użycie zdania *rollback* nic już nie zmieni, gdyż między ostatnim zdaniem *commit* a obecnym *rollback* nic się nie wydarzyło.

Należy pamiętać, że zdania *create...*, *alter....*, *drop....*, *rename.....* generują w niejawnny sposób zatwierdzenie transakcji czyli *commit*.

Dlatego, jeśli w powyższym scenariuszu po realizacji punktu 2, a przed realizacją punktu 3 wykonane zostanie, na przykład, zdanie:

```
create table tmp_tabela
(
    iden varchar2(3) primary key
    ,opis varchar2(100)
);
```

to nastąpi zatwierdzenie kasowania zawartości tabeli **WYPOZYCZENIE** i dalsze działania na podstawie scenariusza nie mają sensu.

Zadania do samodzielnego wykonania:

Działanie na modelu Biblioteka

1. Utworzyć w swoim schemacie tabele modelu relacyjnego *Biblioteka* przy pomocy skryptu lab_BD1_ver2.sql metodą „@c:\temp\lab_BD1_ver2.sql”. Wypełnić tabele przykładowymi danymi wykorzystując wszystkie postacie zdanie *insert*. Układ danych powinien zapewnić obserwację relacji 1:N czyli jedna książka była wypożyczona wiele razy oraz jeden czytelnik kilka razy przychodził do biblioteki w celu wypożyczenia książek.
2. Na tak skonfigurowanym modelu z danymi dokonać modyfikacji struktury poprzez zapewnienie poprawności rejestrowania wypożyczeń. Struktura tabeli **WYPOZYCZENIA** dopuszcza możliwość rejestrowania transakcji bez określenia numeru wypożyczonej książki. Należy to zmienić.
3. Dokonać modyfikacji danych w dwóch wariantach. Pierwszy - modyfikacji podlega jeden wiersz (modyfikacja w oparciu o klucz główny), a drugi - modyfikacji podlega zbiór wierszy, ale nie wszystkie.
4. Zaprezentować skuteczność lub nie kasowania wiersza nadziedzkiego w dwóch wariantach. Pierwszy - wiersz nadziedzony jest w relacji z wierszami podrzędnymi, a drugi - nie jest.

Tworzenie nowego modelu

1. Zaprojektować prosty model bazy danych składający się z trzech tabel i odzwierciedlający ewidencję studentów uczelni z uwzględnieniem podziału na rodzaj studiowania i kierunki studiów.

Tabela BD1_Student – zawierająca dane studenta,

Tabela BD1_Kierunki_Studiow – zawierająca nazwy kierunków
(Informatyka, Psychologia,.....),

Tabela BD1_Rodzaj_Studiow – zawierająca nazwy sposobów studiowania
(Dzienne, Zaoczne, Indywidualne, Podyplomowe, MBA).

Tabele BD1_Kierunki_Studiow oraz BD1_Rodzaj_Studiow są klasycznymi tabelami słownikowymi. Na ogół struktura takich tabel zawiera dwie kolumny: kod i opis:

```
(  
kod varchar2(3) primary key  
,opis varchar2(100)  
);
```

Tabele takie często są używane w systemach bazodanowych do przechowywania danych stałych, na przykład mogą to być zbiory nazw województw, nazw walut, komórek organizacyjnych itp. Kolumna stanowiąca klucz główny może być typu varchar2 (ale zdecydowanie krótsza niż kolumna stanowiąca opis obiektu) lub typu numerycznego.

Tabelę BD1_Student należy traktować jako tabelę dynamiczną (transakcyjną).

2. Zdefiniować klucze główne oraz klucze obce określające relacje między tabelami:

Relacje: student studiuje na określonym kierunku,
student studiuje określonym sposobem studiowania.

3. Na tak zbudowanym modelu przećwiczyć wprowadzanie danych do tabel, ich modyfikację oraz usuwanie na różne sposoby (zdania *insert*, *update*, *delete*).
4. Przećwiczyć skuteczność działania klucza obcego w celu utrzymania spójności bazy danych (np. poprzez wprowadzenie do ewidencji studenta, który studiuje na kierunku, którego brak w ewidencji kierunków oraz skasowanie z tabeli kierunków studiów pozycji, w przypadku gdy na ten kierunek są zapisani studenci).
5. Zmodyfikować model bazy danych poprzez wprowadzenie do jednej z tabel nowej kolumny (np. do tabeli kierunków studiów kolumnę określającą datę uruchomienia danego kierunku) i uzupełnić tę kolumnę danymi (zrobić to na dwa sposoby: wszystkie wiersze na raz oraz wybiórczo).
6. Sprawdzić czy jest możliwość zmiany struktury tabeli poprzez zmianę wielkości wybranej kolumny w przypadku, gdy jest ona wypełniona (np. kolumna Nazwisko w tabeli BD1_Student). Należy zwiększyć rozmiar kolumny np. do 100 i zmniejszyć np. do 5.
7. Utworzyć dwa skrypty. Pierwszy (np. *create_BD1.sql*) zawierający zdania tworzące obiekty bazodanowe i zdania wprowadzające testowe dane do tabel oraz drugi (np. *drop_BD1.sql*) usuwający wszystkie tabele opracowanego modelu bazy danych.
8. Usunąć jedną z referencji między tabelami i przetestować tego skutki poprzez wprowadzanie danych umożliwiających osiągnięcie przez bazę danych stanu niespójności. Odpowiednimi zdaniami SQL doprowadzić z powrotem do spójności bazy danych.

Bazy Danych laboratorium

**Laboratorium
BD2**

Pozyskiwanie informacji z jednej tabeli - opis przykładu

Tabela BD2_ZBIORCZA zawiera ewidencję wyników cyklu biegowych zawodów sportowych. W ramach jednego biegu zapisywana jest klasyfikacja zawodników, na którą składają się ewidencyjne dane zawodnika, punkty zdobyte w klasyfikacji generalnej i punkty zdobyte w klasyfikacji wiekowej.

Ewidencyjne dane zawodnika to numer ewidencyjny, imię i nazwisko, płeć, rok urodzenia, kategoria wiekowa, przynależność klubowa w postaci numeru klubu i jego nazwy. Zawodnicy są podzieleni na kategorie wiekowe według roku urodzenia, np. kategoria IV to mężczyźni urodzeni między 1957 i 1967 rokiem, a K-II to kobiety urodzone między 1977 i 1986 rokiem.

Punkty klasyfikacji generalnej – pierwszych pięćdziesięciu mężczyzn bez względu na wiek otrzymuje punkty według zasady: 1 miejsce – 50 pkt, 2 miejsce – 49 pkt, ……, 50 miejsce – 1 pkt, pozostały – pole nie jest wypełnione. Analogicznie jest wśród kobiet.

Punkty w klasyfikacji wiekowej (w kategoriach) – pierwszych pięćdziesięciu zawodników w danej kategorii wiekowej (np. II) otrzymuje punkty według podobnej zasady jak w klasyfikacji generalnej. Czyli w ramach tej klasyfikacji zarówno najlepszy zawodnik z kategorii II otrzymuje 50 pkt, jak również najlepszy zawodnik kategorii V też otrzymuje 50 pkt. Analogicznie jest wśród kobiet.

Na końcu cyklu składającego się z kilku biegów sumuje się punkty zdobyte przez zawodników w poszczególnych biegach i na tej podstawie ustala końcowe klasyfikacje: generalną i w kategoriach dla zawodników i generalną dla klubów powstającą na podstawie sumy punktów zdobytych przez zawodników danego klubu w klasyfikacji generalnej.

Poniżej przedstawiona jest struktura tabeli BD2_ZBIORCZA:

BD2_ZBIORCZA			
nr_zawodow	NUMBER(2)	<pk>	not null
nr_zawodnika	NUMBER(4)	<pk>	not null
imie	VARCHAR2(15)		null
nazwisko	VARCHAR2(30)		not null
plec	VARCHAR2(1)		not null
rok_urodzenia	NUMBER(4)		null
nr_klubu	NUMBER(3)		not null
klub	VARCHAR2(40)		not null
kategoria	VARCHAR2(6)		null
pkt_generalna	NUMBER(2)		null
pkt_kategorie	NUMBER(2)		null

Zdania tworzące tę tabelę zawarte są w skrypcie *lab_BD2_tab.sql*. Definicje kolumn *imie*, *rok_urodzenia* i *kategoria* dopuszczają wartości *null*. Nie jest to w pełni zgodne z rzeczywistością, ale zostało tak przygotowane w celu wykonania serii ćwiczeń na wartościach *null*.

Porównując skrypt *lab_BD2_tab.sql* z diagramem tabeli można zauważyc dodatkowe definicje (*constraint*) typu *check* służące do nadawania pewnych ograniczeń na wartości umieszczane w wybranych kolumnach. Kolumny *pkt_generalna* i *pkt_kategorie* mogą zawierać tylko wartości z przedziału [1..50] lub *null* - co wynika z regulaminu zawodów, a kolumna *plec* - jednoliterowe kody płci.

Warto zwrócić uwagę na złożony klucz główny. Każdy zawodnik może startować w kilku zawodach w sezonie i dlatego, aby jednoznacznie określić, o który wynik chodzi przy przeszukiwaniu bazy danych trzeba podać numer zawodów i numer zawodnika. I dopiero ta para wartości jednoznacznie wskaże wiersz w tabeli. Dodatkowo istotny jest sposób definiowana złożonego klucza głównego. Jeden ze sposobów został zaprezentowany w skrypcie *lab_BD2_tab.sql*, a drugim jest użycie zdania SQL *alter table....add constraint....primary key....*

Import danych do tabeli z pliku płaskiego typu wycinek (csv) przy użyciu Oracle SQL Developer

Do ładowania danych do tabel bazodanowych w środowisku Oracle używa się kilku narzędzi lub metod. Do najpopularniejszych należą SQL Loader i Oracle Data Pump, które w tym materiale nie będą omawiane. Szczegółowe informacje można znaleźć w dokumentacji Oracle.

Zostanie, natomiast, omówiona metoda wypełniania tabel danymi przy użyciu Oracle SQL Developera i wbudowanej funkcjonalności Import Data.

Po założeniu tabeli (skrypt *lab_BD2_tab.sql*) należy postępować jak poniżej:

Wyświetlić strukturę utworzonej tabeli i poprzez Actions... przejść do importu danych (Import Data...) ustawiając:

- nazwę importowanego pliku: *lab_BD2_dane.csv*
- umieszczenie nagłówka pliku (Header): nie
- format: csv
- kodowanie: windows-1250 (Cp1250)
- separator pól (Delimiter): przecinek,
- znaki ograniczające dane tekstowe (Left i Right Enclosure): apostrof (')
- metodę importu: Insert Script
- wybór kolumn (Choose Columns): pozostawić bez zmian
- mapowanie kolumn (Match By): Position
- definicje kolumn: sprawdzić (ewentualnie zmienić) poprawność mapowania kolumn danych (Source Data Columns i Target Table Columns)

, na koniec obejrzeć szczegóły zdefiniowanych ustawień i zakończyć proces importu.

Zapoznać się z wygenerowanym skryptem umieszczonym automatycznie w Oracle SQL Developer i go wykonać, a następnie zatwierdzić transakcję zdaniem *commit*. Dodatkowo znaleźć w systemie operacyjnym folder: C:\Users\%USER\AppData\Local\Temp oraz plik o nazwie *Import-lab_BD2_dane-csv.....sql* i wyświetlić jego zawartość.

Obejrzeć zawartość tabeli BD2_ZBIORCZA, szczególną uwagę zwrócić na kodowanie polskich znaków. Wygenerowany skrypt zapamiętać jako *lab_BD2_insert.sql*.

Wykonać zdanie SQL:

```
select * from bd2_zbiorcza;
```

W tabeli powinny znajdować się dane:

NR_ZAWODOW	NR_ZAWODNIKA	IMIE	NAZWISKO	PLEC	ROK_URODZENIA	NR_KLUBU	KLUB	KATEGORIA	PKT_GENERALNA	PKT_KATEGORIE
148	4	173 Paulina	Stachurska	K	1989	380 UKS Bielany	K-I	47	50	
149	1	182 Aleksandra	Dobrowolska	K	1975	348 ENTRE.PL Team	K-III	20	44	
150	2	182 Aleksandra	Dobrowolska	K	1975	348 ENTRE.PL Team	K-III	30	49	
151	3	182 Aleksandra	Dobrowolska	K	1975	348 ENTRE.PL Team	K-III	33	47	
152	4	182 Aleksandra	Dobrowolska	K	1975	348 ENTRE.PL Team	K-III	25	40	
153	1	202 Stefan	Karpisz	M	1947	67 Warszawa-Praga Południe	V	(null)	21	
154	1	203 Grzegorz	Stańczyk	M	1955	41 Warszawa-Ursynów	V	(null)	23	
155	1	204 Andrzej	Gromko	M	1957	50 Warszawa-Śródmieście	IV	(null)	22	
156	1	205 Dawid	Żywek	M	1985	400 OKS Start Otwock	II	48	48	

Wykonać drugie zdanie:

```
select count(*) from bd2_zbiorcza;
```

Powinien pojawić się wynik liczbowy wskazujący ile wierszy zostało wpisanych do tabeli:

COUNT(*)
1 838

W przypadku niepowodzenia należy proces ładowania danych powtórzyć.

Zdanie SELECT – podstawy

W swej najprostszej postaci zdanie SELECT wymaga tylko wskazania tabeli, w której znajdują się oczekiwane dane. Ta najprostsza forma wygląda jak poniżej:

```
select * from nazwa_tabeli
```

```
np.: select * from bd2_zbiorcza;
```

Zbiorem wynikowym powyższego zdania będą wszystkie dane zawarte w tabeli czyli wszystkie wiersze i wszystkie kolumny.

Podstawowa składnia tego zdania jest następująca:

```
select kolumn1, kolumn2, ...., kolumna_n from nazwa_tabeli
```

```
np.: select nr_zawodow, imie, nazwisko from bd2_zbiorcza;
```

NR_ZAWODOW	IMIE	NAZWISKO
1	Krystyna	Sawicka
1	Anna	Jasek
1	Ewa	Tkaczyk
2	Ewa	Tkaczyk
4	Ewa	Tkaczyk
2	Wanda	Misiaczek
3	Wanda	Misiaczek
1	Anna	Kolarska
1	Maria	Stańczak
3	Maria	Stańczak
1	Agnieszka	Teichert
	

Uwagi:

- Kolejność nazw kolumn jest dowolna i może być podzbiorem wszystkich kolumn stanowiących strukturę tabeli czyli powyższe zdanie może wyglądać następująco:

```
select imie, nazwisko, nr_zawodow from bd2_zbiorcza;
```

- Zawężenie wyboru kolumn może prowadzić do powtarzalności wierszy powodując nadmiarowość zbioru wynikowego.

Zdanie:

```
select nazwisko, imie from bd2_zbiorcza;
```

da wynik:

NAZWISKO	IMIE
Sawicka	Krystyna
Jasek	Anna
Tkaczyk	Ewa
Tkaczyk	Ewa
Tkaczyk	Ewa
Misiaczek	Wanda
Misiaczek	Wanda
Kolarska	Anna
Stańczak	Maria
Stańczak	Maria

.....

Aby uzyskać unikatową listę nazwisk należy użyć konstrukcji:

```
select distinct nazwisko, imie from bd2_zbiorcza;
```

Wtedy wynik będzie następujący:

NAZWISKO	IMIE
Kolarska	Anna
Teichert	Agnieszka
Gliniewicz	Elżbieta
Zielona	Magdalena
Dobosz	Elżbieta
Hirska	Ela
Markowska	Barbara
Paczkowska	Olga
Butkiewicz	Emilia

.....

Warto zaznaczyć, że klauzula **distinct** nie odnosi się do kolumny, przed którą bezpośrednio stoi, lecz do układu wszystkich kolumn na liście select. Błędne jest zatem zdanie:

```
select distinct nazwisko, distinct imie from bd2_zbiorcza;
```

Filtrowanie danych wynikowych za pomocą klauzuli WHERE

Fraza WHERE umożliwia definiowanie jednego lub wielu warunków, które muszą być spełnione przez każdy wiersz tabeli kwalifikowany do ostatecznego zbioru wynikowego. W ramach tej frazy można używać operatorów porównania ($=$, $<$, $>$, \leq , \geq , \neq), operatorów logicznych (AND, OR, NOT) oraz dodatkowo takich operatorów jak BETWEEN, ANY, SOME, IN oraz LIKE.

Zastosowanie operatorów porównania:

np.

```
select distinct imie, nazwisko, rok_urodzenia
    from bd2_zbiorcza
    where nr_klubu = 10;
```

zwraca dane zawodników z klubu o identyfikatorze 10, którzy chociaż raz wystartowali w cyklu zawodów:

IMIE	NAZWISKO	ROK_URODZENIA
Krzysztof	Mróz	1973
Karol	Maciuszek	1986
Jarosław	Nizak	1970

Zwrócić należy uwagę, że bez klauzuli *distinct* wynik będzie inny:

IMIE	NAZWISKO	ROK_URODZENIA
Karol	Maciuszek	1986
Karol	Maciuszek	1986
Karol	Maciuszek	1986
Krzysztof	Mróz	1973
Jarosław	Nizak	1970

gdyż jeden zawodnik może startować kilka razy w cyklu zawodów.

Zastosowanie operatorów logicznych:

np.

.....
where plec = 'M' and rok_urodzenia > 1980 (iloczyn logiczny)

oznacza wybór wierszy dotyczących mężczyzn urodzonych po 1980 roku,

.....
where plec = 'M' or rok_urodzenia < 1980 (suma logiczna)

oznacza wybór wszystkich wierszy dotyczących mężczyzn oraz dodatkowo kobiet urodzonych przed 1980 rokiem.

Przykład złożonego zdania logicznego:

.....
where not
((nr_klubu < 4 or nr_klubu = 20)
and
 (rok_urodzenia > 1980 or rok_urodzenia <= 1960)
)

Należy zwrócić uwagę na obowiązującą kolejność wykonywania działań z operatorami *and* i *or* oraz konieczność używania nawiasów w celu jej zmiany.¹

Na podstawie prawa de Morgana² można powyższy warunek przekształcić następująco:

.....
where
not (*nr_klubu* < 4 *or* *nr_klubu* = 20)
or
not (*rok_urodzenia* > 1980 *or* *rok_urodzenia* <= 1960)

Szczegółową analizę tego przykładu pozostawia się czytelnikowi. Werbalnie można stwierdzić, że na podstawie drugiego wariantu powyższego przykładu do zbioru wynikowego zostaną zakwalifikowane wiersze, które albo nie spełniają pierwszego warunku (*nr_klubu*) albo nie spełniają drugiego warunku (*rok_urodzenia*). Innymi słowy (na podstawie wariantu pierwszego) do zbioru wynikowego nie zostaną zakwalifikowane wiersze, które spełniają pierwszy i drugi warunek jednocześnie.

Zastosowanie pozostałych operatorów:

Operator **BETWEEN**:

Umożliwia określenie przedziału, do którego muszą należeć akceptowane wartości:
np.:

.....
where *nr_zawodow* *between* 2 *and* 4

jest równoważne frazie:

.....
where *nr_zawodow* >= 2 *and* *nr_zawodow* <= 4

A fraza:

.....
where *nr_zawodow* **not between** 2 *and* 4

lub jej równoważna:

.....
where **not** *nr_zawodow* *between* 2 *and* 4

jest zaprzeczeniem poprzedniej frazy.

Operator **IN**:

Sprawdza, czy wartość wskazanej kolumny odpowiada którejś z wartości wymienionej na liście argumentów tego operatora.

np.:

.....
where *kategoria* **in** ('K-II', 'K-IV', 'K-VI')

¹ Szczególnie jest to istotne w przypadku używania operatora logicznego *or* w połączeniu z operatorem *and*.

² Pierwsze prawo De Morgana - prawo zaprzeczania koniunkcji: *not* (*p and q*) \Leftrightarrow *not p or not q*, gdzie *p* i *q* oznaczają zdania logiczne.

Do zbioru wynikowego zostaną zakwalifikowane wiersze, w których pole kategoria przyjmuje jedną z trzech wartości w nawiasie.

Operator **LIKE**:

Umożliwia stosowanie symboli wieloznacznych w warunkach przeszukujących pola znakowe. Symbol wieloznaczny nie definiuje konkretnego znaku, a jedynie dopasowuje do zdefiniowanego wzorca dowolny znak.

Na ogół stosuje się dwa symbole wieloznaczne:

% -dowolna liczba znaków,
_ -jeden znak,

np.

where nazwisko like 'Ko%'

oznacza wybór tych wierszy tabeli, w których nazwisko zaczyna się na Ko czyli np. Kos lub Kowalski,

ale:

where nazwisko like 'Ko_'

oznacza wybór tych wierszy, w których nazwisko jest trzyliterowe i zaczyna się na Ko czyli np. Kot, Kos, ale nie Kowalski,

a np.:

where imie like '%ś%'

oznacza wybór wierszy, w których imię zawiera w sobie literę 'ś'.

Zarządzanie wartościami **NULL**

Jak już wcześniej wspomniano, w przypadku, gdy do jakiegoś pola w tabeli nie wprowadzono określonej wartości to pole to ma ustawioną wartość *null*.

W przypadku wprowadzania wiersza do tabeli zdaniem *insert* można to osiągnąć w dwojakim sposobie. Pierwszy polega na wyspecyfikowaniu wszystkich kolumn w tabeli a na pozycjach wartości w odpowiednim miejscu wpisanie *null* lub nie wpisanie niczego:

```
insert into tabela (kol1, kol2, kol3)  
values (wartość_1, null, wartość_3);
```

lub

```
insert into tabela (kol1, kol2, kol3)  
values (wartość_1, ' ', wartość_3); -- pusty ciąg znaków na pozycji drugiej
```

Natomiast zdanie:

```
insert into tabela (kol1, kol2, kol3)  
values (wartość_1, ' ', wartość_3); -- spacja na pozycji drugiej
```

wprowadza do kolumny spację a nie *null*.

Drugi sposób polega na uwzględnieniu w specyfikacji zdania *insert* tylko tych kolumn, do których wprowadza się dane:

```
insert into tabela (kol1, kol3)
values    (wartość_1, wartość_3); -- kol2 istnieje w strukturze tabeli
```

Należy mieć na uwadze fakt, że realizacja powyższego zdania jest możliwa tylko pod warunkiem, że w definicji tabeli dopuszcza się możliwość występowania *null* w kol2 czyli dla tej kolumny nie występuje *not null*.

Chcąc ustawić jakieś pole lub kilka pól tej samej kolumny na wartość *null* można to zrobić na przykład tak:

```
update tabela
set kol2 = null, kol4 = null
where kol1 like 'A%';
```

Null posiada następujące właściwości:

- null <> dowolna_określona_wartość
- null not > dowolna_określona_wartość
- null not < dowolna_określona_wartość
- null <> null
- null + dowolna_określona_wartość = null
- null and true = null
- null and false = false
- null or true = true
- null or false = null

Chcąc skasować określone wiersze w tabeli należy pamiętać o podstawowej własności *null* wyseznególnionej powyżej na pierwszym miejscu czyli *null* nie równa się "niczemu".

Zatem zdanie:

```
delete tabela
where kol2 = null;
```

nie skasuje żadnego wiersza w tabeli, pomimo faktu, że w kol2 znajdują się wartości *null*. Kasowanie będzie skuteczne, gdy zamiast znaku równości użyte zostanie słowo *is*.

```
delete tabela
where kol2 is null;
```

Należy zatem pamiętać, aby przy przeszukiwaniu tabeli zdaniem *select* we frazie *where* również używać filtru ...**kolumna is null**, a nie ...*kolumna = null*.

Porządkowanie zbioru wynikowego przy pomocy frazy ORDER BY:

Standardowo zbiór wynikowy nie jest uporządkowany w żaden sposób. O kolejności wierszy decyduje system zarządzania bazą danych (SZBD). Na ogół jest zgodny z kolejnością wpisywania wierszy do tabeli (gdy brak jest zdefiniowanego klucza głównego) lub zgodnie z kluczem głównym. W przypadku, gdy zachodzi potrzeba uporządkowania zbioru według zadanych kryteriów należy użyć frazy *order by*, którą umieszcza się na końcu zdania *select*.

np. zdanie:

```
select *
from bd2_zbiorcza
where nr_zawodow = 1
order by pkt_generalna;
```

zwróci zbiór wyników zawodów o identyfikatorze 1 posortowanych rosnąco (od 1 do 50 pkt) według zdobytych punktów. Należy zwrócić uwagę, że wiersze zawierające *null* w kolumnie *pkt_generalna* są umieszczone na końcu zbioru.

Chcąc umieścić wiersze z wartościami NULL w kolumnie klucza sortowania na początku zbioru wyników należy użyć konstrukcji:

.....
order by *pkt_generalna* *nulls first*;

Domyślnie ustawiony jest rosnący kierunek sortowania (*ascending* – skrót *asc*). W przypadku zmiany kierunku sortowania na malejący należy użyć klauzuli *desc* (*descending*), np.:

.....
order by *pkt_generalna* *desc*;

Uwagi końcowe:

1. Istnieje możliwość łączenia ze sobą kilku kolumn znakowych w zdaniu *select*. Na przykład chcemy przedstawić imię i nazwisko pisane łącznie ze standardowym odstępem w postaci jednej spacji. W takim przypadku można skorzystać z operatora konkatenacji (łączenia ciągów znakowych) w sposób podany poniżej:

```
select distinct nazwisko || ' ' || imie, rok_urodzenia
from bd2_zbiorcza
order by rok_urodzenia desc;
```

NAZWISKO " IMIE	ROK_URODZENIA
Dawidzka Katarzyna	1999
Bielinski Marcin	1993
Andres Anna	1992
Cygler Tomasz	1992
Banaszek Marta	1991
Kuczkowski Mateusz	1991
Chorążewicz Bartosz	1990
Fitzgibbon Sławomir	1990
Gajda Monika	1990
Jarosiewicz Bartek	1990

.....

2. W zdaniu *select* można zmieniać nazwy kolumn w zbiorze wyników. Na przykład łatwo zauważyc, że w powyższym przykładzie nazwa kolumny nie jest zbyt atrakcyjna. Wystarczy zastosować alias jak poniżej:

```
select distinct nazwisko || ' ' || imie as Nazwisko, rok_urodzenia as Rocznik
from bd2_zbiorcza
order by rok_urodzenia desc;
```

3. Nazwa kolumny w zbiorze wynikowym może składać się z kilku wyrazów oddzielonych spacjami. W takim przypadku należy pamiętać o użyciu cudzysłów, jak w poniższym przykładzie (ale nie apostrofów):

```
select distinct nazwisko || ' ' || imie as "Nazwisko i imię", rok_urodzenia as "Rocznik"
from bd2_zbiorcza
order by rok_urodzenia desc;
```

Nazwisko i imię	Rocznik
Dawidzka Katarzyna	1999
Bielinski Marcin	1993
Andres Anna	1992
Cygler Tomasz	1992
Banaszek Marta	1991
Kuczkowski Mateusz	1991
Chorążejewicz Bartosz	1990
Fitzgibbon Sławomir	1990
Gajda Monika	1990
Jarosiewicz Bartek	1990

.....

4. Słowo *as* w definicji aliasu można pominąć.
5. Aby posortować zbiór wyników według kilku kluczów sortowania należy użyć poniższej konstrukcji frazy *order by*:

.....
order by pierwszy_klucz_sortowania [desc], drugi_klucz_sortowania [desc],.....

, przy czym kierunek sortowania należy ustalić dla każdego klucza oddzielnie.

6. We frazie *order by* jako klucze sortowania umieszcza się, na ogólnie, kolumny, które równocześnie występują we frazie *select*, na przykład:

```
select nazwisko, imie, rok_urodzenia
from bd2_zbiorcza
order by rok_urodzenia;
```

Możliwa jest inna konstrukcja polegająca na sortowaniu zbioru wyników według kolumny nie występującej na liście *select*, na przykład:

```
select nazwisko, imie
from bd2_zbiorcza
order by rok_urodzenia;
```

Ale powyższa konstrukcja nie jest możliwa w przypadku użycia klauzuli *distinct*, na przykład:

```
select distinct nazwisko, imie
from bd2_zbiorcza
order by rok_urodzenia;
```

Jednym ze sposobów rozwiązania tego problemu jest użycie funkcji agregującej³, np.:

```
select nazwisko, imie
from bd2_zbiorcza
group by nazwisko, imie
order by max(rok_urodzenia);
```

Klauzulę *distinct* można zastąpić odpowiednio skonstruowaną frazą *group by*:

<pre>select distinct nazwisko, imie from bd2_zbiorcza;</pre>	=	<pre>select nazwisko, imie from bd2_zbiorcza group by nazwisko, imie;</pre>
---------------------------------------------------------------------	----------	-----------------------------------------------------------------------------

³ Funkcje agregujące zostaną omówione w Laboratorium BD4.

Zadania do samodzielnego wykonania

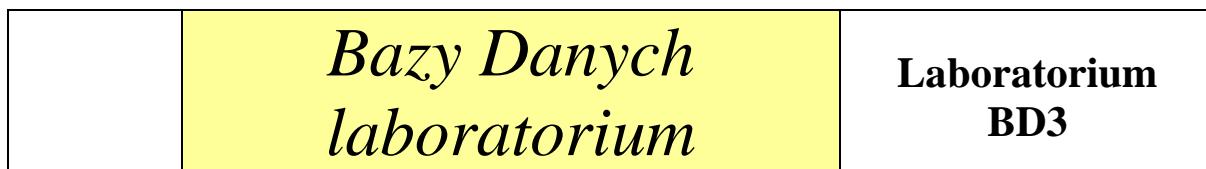
Działania na tabeli BD2_ZBIORCZA_TEMP:

1. Zmodyfikować plik lab_BD2_dane.csv poprzez wprowadzenie na początku nazw kolumn (zgodnych z nazwami kolumn w tabeli lub też nie) i przeprowadzić proces ładowania danych do tabeli BD2_ZBIORCZA_TEMP, która ma mieć dokładnie taką samą strukturę jak tabela BD2_ZBIORCZA.
2. W tabeli BD2_ZBIORCZA_TEMP zmienić kolumnę *plec* tak, aby zawierała wartości odpowiednio *Kobieta* lub *Mężczyzna* zamiast dotychczasowych *K* i *M*, a nazwę kolumny zmienić na *plec_nazwa*. Po przeprowadzeniu niezbędnych modyfikacji zdaniem *update* przetestować ich skuteczność (w kolumnie określającej płeć mogą być tylko wartości *Kobieta* lub *Mężczyzna*).
3. W tabeli BD2_ZBIORCZA_TEMP zmodyfikować kolumnę *pkt_kategorie* tak, aby konieczne było wprowadzenie do niej wartości różnej od null (czyli, aby otrzymała atrybut not null). W miejscach dotychczas występujących wartości null wstawić 0.

Działania na tabeli BD2_ZBIORCZA:

Zaprojektować zdania SQL generujące zbiory wynikowe przy następujących założeniach:

1. Mężczyźni urodzeni po 1975 roku i należący do klubów o numerach między 3 i 10. Uporządkowanie – według numerów klubów i roczników (malejąco).
2. Zawodnicy, których nazwisko kończy się na 'ski' lub 'ska' należący do jednej z kategorii (I, II, K-II, K-V). Uporządkowanie – na początku kobiety, a potem mężczyźni, a w ramach płci alfabetycznie.
3. Zawodnicy należący do klubu 'KB Gymnasion Warszawa', którzy nie zdobyli punktów w klasyfikacji generalnej. Uporządkowanie - według kategorii wiekowej i nazwisk.
4. Zawodnicy, którzy nie zdobyli punktów w klasyfikacji kategoriami i należą do kategorii (II, III, V). Uporządkowanie – według kategorii, klubów i aliasu stanowiącego połączenie Nazwiska i imienia.



Pozyskiwanie informacji z wielu tabel - opis przykładu

Model bazy danych, na podstawie którego będą realizowane to i dalsze laboratoria stanowi rozwinięcie przykładu poprzedniego (Laboratorium BD2). Ten model zawiera kilka tabel połączonych relacjami ze sobą, ale istota problemu pozostaje ta sama. Poniżej przedstawiony jest schemat relacyjnego modelu zobrazowany przy pomocy aplikacji DBeaver w notacji IDEFIX:

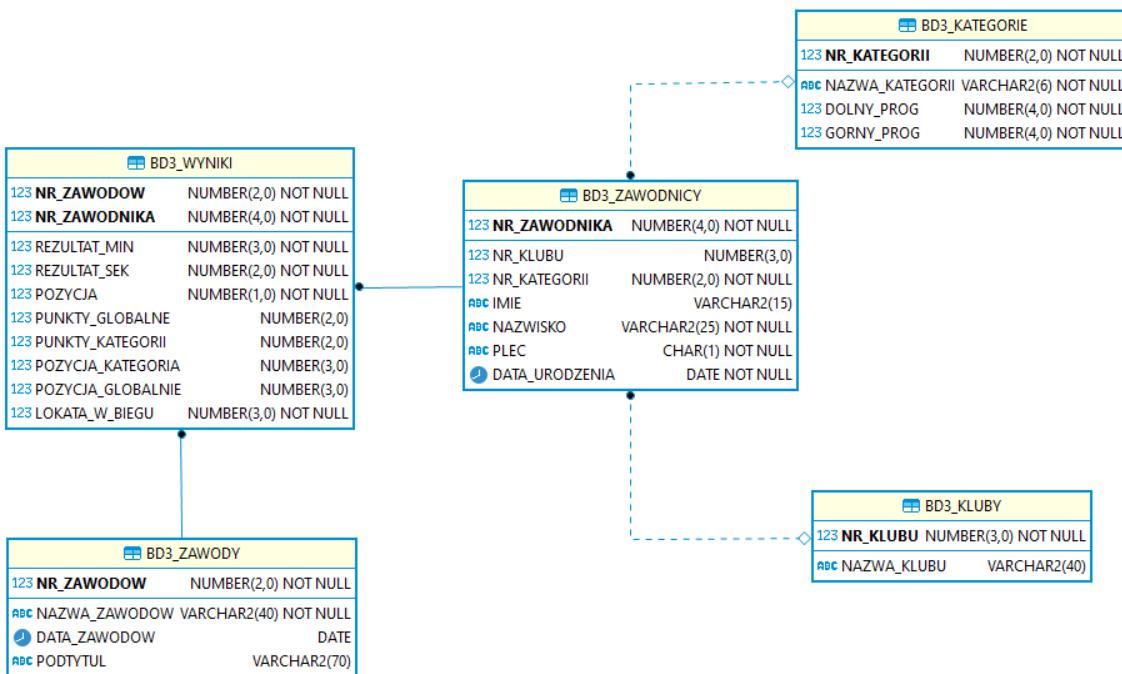


Tabela BD3_WYNIKI zawiera ewidencję wyników cyklu biegowych zawodów sportowych. W ramach jednego biegu zapisywana jest klasyfikacja zawodników, na którą składają się wyniki sportowe zawodnika, punkty zdobyte w klasyfikacji generalnej i punkty zdobyte w klasyfikacji wiekowej.

Tabela BD3_ZAWODNICY to numer ewidencyjny, imię i nazwisko, płeć, data urodzenia. Kategoria wiekowa i przynależność klubowa zawodnika realizowana jest w relacji z odpowiednimi tabelami.

Zawodnicy są podzieleni na kategorie wiekowe (tabela BD3_KATEGORIE) według roku urodzenia, np. kategoria IV to mężczyźni urodzeni między 1957 (dolny próg) i 1967 (górnny próg) rokiem, a K-II to kobiety urodzone między 1977 i 1986 rokiem.

Punkty klasyfikacji generalnej (*Punkty globalne* w tabeli BD3_WYNIKI) – pierwszych pięćdziesięciu mężczyzn bez względu na wiek otrzymuje punkty według zasady: 1 miejsce – 50 pkt, 2 miejsce – 49 pkt, ……, 50 miejsce – 1 pkt, pozostały – pole jest niewypełnione. Analogicznie jest wśród kobiet.

Punkty w klasyfikacji wiekowej (*Punkty kategorii* w tabeli BD3_WYNIKI) – pierwszych pięćdziesięciu zawodników w danej kategorii wiekowej (np. II) otrzymuje punkty według podobnej zasady jak w klasyfikacji generalnej. Czyli w ramach tej klasyfikacji zarówno najlepszy zawodnik z kategorii II otrzymuje 50 pkt, jak również najlepszy zawodnik kategorii V też otrzymuje 50 pkt. Analogicznie jest wśród kobiet.

Na końcu cyklu składającego się z kilku biegów sumuje się punkty zdobyte przez zawodników w poszczególnych biegach i na tej podstawie ustala końcowe klasyfikacje: generalną i w kategoriach dla zawodników i generalną dla klubów powstającą na podstawie sumy punktów zdobytych przez zawodników danego klubu w klasyfikacji generalnej.

W skrypcie *lab_BD3_create.sql* można zapoznać się dokładnie ze zdaniami SQL implementującymi powyższy model w bazie danych.¹

W powyższym skrypcie występują dodatkowe (poza definicją kluczy głównych i obcych) ograniczenia na wybrane kolumny:

1. Klauzula *not null* – system zarządzania bazą danych (SZBD) nie dopuści do wpisania wiersza do tabeli, w którym kolumna posiadająca tę klauzulę nie będzie miała określonej wartości,
2. Klauzula *check* – ogranicza wartości w danej kolumnie do wyspecyfikowanych, np. ograniczenie *Plec in ('M', 'K')* zapewnia, że w kolumnie *Plec* może wystąpić tylko litera M lub K (lecz nie 'm' lub 'k'),
3. Klauzula *default* - definiuje domyślną wartość w kolumnie.

Implementacja modelu w schemacie Oracle

Implementacja modelu będzie podobna do tej przeprowadzonej w Laboratorium BD2, przy czym tym razem nie będzie to jedna tabela tylko pięć połączonych ze sobą relacjami w postaci kluczy obcych. W takim przypadku może być istotna kolejność zarówno tworzenia tabel, jak i wypełniania ich danymi. Analizując graficzną prezentację modelu można zauważać hierarchię obiektów. Na przykład relacja między tabelami BD3_KATEGORIE i BD3_ZAWODNICY określa tę pierwszą tabelę jako obiekt nadzędny w stosunku do drugiej. To zawodnicy są "podpięci" pod daną kategorię wiekową (do jednej kategorii należy wielu zawodników). W zdaniach SQL *create table....lub alter table....* tworzących referencje objawia się to tym, że definicja referencji (klucza obcego) znajduje się w tabeli podrzędnej. Klucz obcy wiążący tabele BD3_KATEGORIE i BD3_ZAWODNICY znajduje się w tabeli drugiej czyli kolejność implementacji tabel podlega zasadzie: "najpierw nadzędny, potem podrzędny".

```
create table BD3_KATEGORIE
(
    NR_KATEGORII      NUMBER(2) primary key,
    NAZWA_KATEGORII   VARCHAR2(6) not null,
    DOLNY_PROG        NUMBER(4) not null,
    GORNY_PROG        NUMBER(4) not null,
);

create table BD3_ZAWODNICY
(
    NR_ZAWODNIKA      NUMBER(4) primary key,
    NR_KLUBU           NUMBER(3) references BD3_KLUBY (NR_KLUBU),
    NR_KATEGORII       NUMBER(2) not null references BD3_KATEGORIE (NR_KATEGORII),
    IMIE               VARCHAR2(15),
    NAZWISKO           VARCHAR2(25) not null,
    PLEC               CHAR(1)      not null,
    DATA_URODZENIA    DATE         not null,
);
```

Niedogodnością tego rozwiązania jest konieczność odpowiedniego uszeregowania tworzonych obiektów, jak również kolejność wprowadzania danych z plików zewnętrznych. Problem ten występuje wyraźnie w modelach zawierających kilkadziesiąt lub więcej tabel.

¹ Dodatkowo w materiałach do laboratorium znajduje się skrypt *lab_BD3_drop.sql*, przy pomocy którego można całkowicie usunąć model BiegiBaza ze swojego schematu.

Drugim wariantem implementacji jest opóźnienie definiowania referencji przy pomocy zdań *alter table...*, które następuje po wszystkich zdaniach *create table....*, w których nie ma definicji referencji. W takim przypadku możliwe jest uszeregowanie tworzonych tabel według alfabetu, następnie wprowadzenie danych do tabel nie analizując skutków działania referencji i na końcu zdefiniowanie referencji. Skrypt *lab_BD3_create.sql* umożliwia implementację modelu według tego wariantu poprzez uruchamianie go partiami.

Trzeci wariant polega na zmianie kolejności wykonywanych czynności według scenariusza:

- tworzenie tabel (zdania *create table...* według alfabetu),
- tworzenie referencji (zdania *alter table...* w dowolnej kolejności),
- wprowadzanie danych do tabel (w odpowiedniej kolejności wynikającej z hierarchii tabel).

Skrypt *lab_BD3_create.sql* umożliwia implementację modelu według trzeciego wariantu i tak należy ją zrealizować metodą „@c:\temp\lab_BD3_create.sql”. Zestawy danych dotyczące każdej z tabel znajdują się w osobnych plikach typu wycinek (csv):

lab_bd3_kategorie.csv
lab_bd3_kluby.csv
lab_bd3_wyniki.csv
lab_bd3_zawodnicy.csv
lab_bd3_zawody.csv

Sposób implementacji jest analogiczny do implementacji tabeli BD2_ZBIORCZA wykonanej w ramach Laboratorium BD2. Jedyną różnicą jest istnienie w plikach z danymi nagłówków z nazwami kolumn tożsamymi z nazwami kolumn w tabelach. Należy to zaznaczyć w momencie importowania danych ustawiając opcję *Header* na Yes.

Po wykonaniu wszystkich czynności należy kontrolnie sprawdzić liczbę wierszy w każdej z tabel zdaniem *select count (*) from tabela*. Wyniki powinny być takie, jak poniżej:²

Tabela	Liczba wierszy
BD3_KATEGORIE	22
BD3_KLUBY	27
BD3_WYNIKI	838
BD3_ZAWODNICY	771
BD3_ZAWODY	4

Zakończyć implementację zatwierdzeniem transakcji (*commit*).

Wyprowadzanie danych z tabel do plików zewnętrznych lub skryptów INSERT przy pomocy Oracle SQL Developer

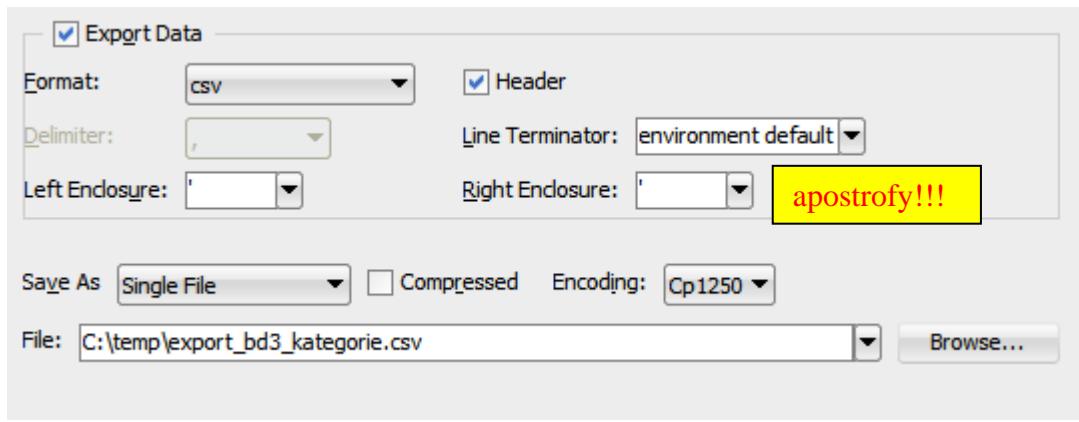
Mając tabelę bazowaną z umieszczonymi w niej danymi można dokonać eksportu, zarówno struktury tabeli (metadanych), jak i danych. Wykorzystuje się funkcję Export... usytuowaną pod przyciskiem Actions... po wyświetleniu struktury tabeli.

Chcąc dokonać eksportu danych z tabeli BD3_KATEGORIE należy:

1. Wybrać tabelę BD3_KATEGORIE w celu zobrazowania jej struktury,
2. Spod przycisku Actions... uruchomić funkcję Export...,
3. Usunąć wybór Export DDL (eksport metadanych),

² W materiałach do laboratorium znajduje się skrypt *lab_BD3_check.sql* zawierający przykładowe zdanie SQL.

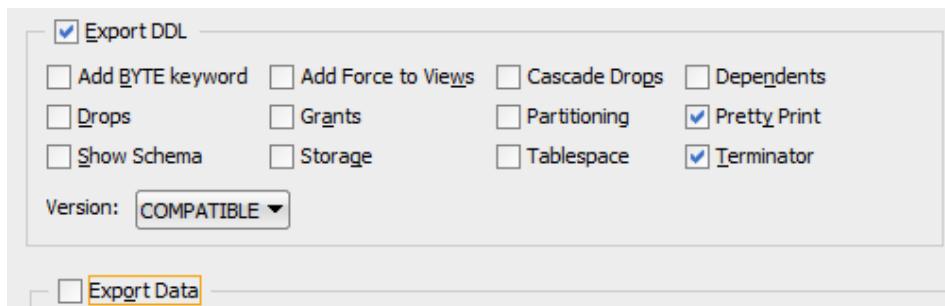
4. Opcje dotyczące eksportu danych ustawić na przykład tak, jak na poniższym rysunku:



5. Wyspecyfikować wszystkie wiersze i wszystkie kolumny tabeli (sekcja Specify Data),
 6. Zakończyć proces eksportu,
 7. Zapoznać się z wygenerowanym plikiem z danymi.

Powtórzyć powyższy scenariusz zmieniając Format na insert oraz nazwę pliku docelowego na *export_bd3_kategorie_insert.sql*. Wygenerowany zestaw zdań *insert* pojawi się w panelu SQL Developera, ale nie należy go wykonywać, tylko zapoznać się z utworzonym skryptem o zadanej nazwie oraz opcjonalnie zapamiętać.

Chcąc dokonać eksportu metadanych czyli struktury tabeli należy uaktywnić wybór Export DDL (ale dezaktywować podzielne opcje według poniższego wzorca) i dezaktywować Export Data:



Wyeksportować metadane do pliku *export_bd3_kategorie_ddl.sql*. Obejrzeć i przeanalizować powstały skrypt. Zaleca się samodzielnie przećwiczyć efekty wyboru opcji Show Schema oraz Drops, jak również równoczesnego eksportu metadanych i danych w postaci zdań *insert*.

Tworzenie nowej tabeli z danymi na podstawie istniejącej tabeli

Przy pomocy języka SQL można tworzyć nową tabelę na podstawie już istniejącej i dodatkowo wypełnić ją danymi. Czynność ta jest bardzo często wykorzystywana przez administratorów baz danych lub projektantów w celu, na przykład, zabezpieczania swoich danych przed skutkami różnych eksperymentów służących testowaniu oprogramowania.

Pierwszy wariant:

Tabela źródłowa (*source*) istnieje i zawiera dane, a tabeli docelowej (*target*) nie ma.

W takim przypadku obowiązuje konstrukcja:

```
create table bd3_kluby_kopia as
select * from bd3_kluby;
```

Odpitanie tabeli BD3_KLUBY_KOPIA zdaniem `select` da wynik:

NR_KLUBU	NAZWA_KLUBU
37	KU AZS WAT Warszawa
1	Allianz Warszawa
2	KB Orientuz Warszawa
3	KB Gymnasion Warszawa
4	Legia Warszawa
9	Grunwald Poznań
6	KB Promyk Ciechanów
7	KB Pułaski Strong Warka

....

Porównując metadane obu tabel (źródłowej i docelowej) można zauważyc jedną istotną różnicę.
Utworzona tą metodą tabela nie ma klucza głównego.

Sekcja Constraints dla tabeli źródłowej:

CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION
PK_BD3_KLUBY	Primary_Key	(null)
SYS_C00299042	Check	"NR_KLUBU" IS NOT NULL

Sekcja Constraints dla tabeli docelowej:

CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION
SYS_C00299437	Check	"NR_KLUBU" IS NOT NULL

Chcąc, aby tabela docelowa miała klucz główny należy go oddziennie zdefiniować przy pomocy zdania `alter table...`.

Uwagi:

1. Tabela docelowa nie dziedziczy również definicji kluczy obcych oraz ograniczeń typu `check` innych niż `not null`.
2. W konstrukcji `create table tabela_kopia as select * from tabela`, zdanie `select` może być dowolnej dopuszczalnej postaci, np.:

```
create table bd3_kluby_warszawa as
select nazwa_klubu from bd3_kluby
where nazwa_klubu like '%Warszawa%';
```

3. Chcąc utworzyć tabelę docelową o takiej samej strukturze jak tabela źródłowa, ale bez danych należy użyć konstrukcji:

```
create table bd3_kluby_kopia as
select * from bd3_kluby
where 1=0;
```

Zdanie logiczne `1=0` nie jest prawdziwe dla żadnego wiersza w tabeli, więc zbiór wyników będzie zbiorem pustym.

Drugi wariant:

Tabela źródłowa (*source*) zawierająca dane oraz tabela docelowa (*target*) istnieją.
W takim przypadku obowiązuje konstrukcja:

```
insert into bd3_zawodnicy_kobiety
select * from bd3_zawodnicy
where plec= 'K';
```

Sposób utworzenia tabeli docelowej jest dowolny:

- można tabelę utworzyć manualnie zdaniem *create table...i alter table....,*
- można wykorzystać omówioną powyżej konstrukcję *create table... as select * from tabela,*
- można wykorzystać funkcję eksportu metadanych poprzez funkcjonalność *Actions.../Export...*

Równozłączenie (złączenie wewnętrzne)

W swojej najczęściej wykorzystywanej postaci złączenie polega na utworzeniu nowych wierszy w zbiorze wynikowym na podstawie odpowiadających sobie wierszy z tabel składowych. Najczęściej to złączenie realizowane jest poprzez klucz główny jednej tabeli i odpowiadający mu klucz obcy w drugiej. W zbiorze wynikowym znajdą się wiersze, dla których w obu tabelach istnieją te same wartości kolumn stanowiących relację, a nie znajdzie się na przykład wiersz, dla którego w jednej tabeli istnieje klucz główny, ale brak go w odpowiadającym mu kluczu obcy drugiej tabeli. Ta cecha klasyfikuje opisywany proces jako *złączenie wewnętrzne (inner join lub join).*

Składnia złączenia wewnętrznego dla dwóch tabel:

```
select kolumny z tabel
from tabela1, tabela2
where tabela1.kolumna_n = tabela2.kolumna_n
```

np.:

```
select imie, nazwisko, nazwa_klubu
from bd3_zawodnicy, bd3_kluby
where bd3_zawodnicy.nr_klubu = bd3_kluby.nr_klubu;
```

IMIE	NAZWISKO	NAZWA_KLUBU
757 Michał	Lenuszyński	KB Promyk Ciechanów
758 Marek	Herman-Iżycki	KB Promyk Ciechanów
759 Piotr	Wojtyński	KB Promyk Ciechanów
760 Piotr	Bobrowski	KB Lechici Zielonka
761 Grzegorz	Protas	KB Lechici Zielonka
762 Przemysław	Augustyniak	Akvedukt Kielce
763 Mikołaj	Jakowicz	KB Trucht Warszawa
764 Tomasz	Zajkowski	KB Trucht Warszawa
765 Jakub	Bednarczyk	KB Trucht Warszawa
766 Leszek	Kuczkowski	KB Lotos Jabłonna

Na powyższym rysunku pokazano kilka ostatnich wierszy zbioru wyników. Ostatni wiersz ma numer 766, co oznacza, że tyle wierszy zwróciło tak skonstruowane zdanie *select*. Porównując tę liczbę z liczbą wierszy w tabeli BD3_ZAWODNICY wykazaną po implementacji modelu i wprowadzeniu danych do tabel można zauważyc różnicę.

Liczność tabeli BD3_ZAWODNICY wynosi 771, a powyższy *select* zwrócił 766 wierszy. Pięciu zawodników nie jest wykazanych w tym zestawie danych. Wynika to z istoty równozłączenia, która polega na tym, że dla pięciu wierszy z tabeli BD3_ZAWODNICY nie można było dobrać odpowiedniej nazwy klubu.

Wniosek: Pięciu zawodników nie ma określonego klubu, do którego należy (w tabeli BD3_ZAWODNICY kolumna nr_klubu zawiera wartości *null*):

Można to sprawdzić zdaniem:

```
select *
from bd3_zawodnicy
where nr_klubu is null;
```

	NR_ZAWODNIKA	NR_KLUBU	NR_KATEGORII	IMIE	NAZWISKO	PLEC	DATA_URODZENIA
1	597	(null)		6 Krzysztof	Misiejuk	M	56/12/25
2	648	(null)		6 Adam	Kostecki	M	53/04/22
3	94	(null)		22 Katarzyna	Wojciuk	K	79/04/13
4	156	(null)		23 Justyna	Małkiewicz	K	76/01/27
5	228	(null)		6 Dawid	Wereda	M	53/10/12

Istnieje druga, równoznaczna, składnia równozłączenia:

```
select imie, nazwisko, nazwa_klubu
      from bd3_zawodnicy
            join bd3_kluby on bd3_zawodnicy.nr_klubu = bd3_kluby.nr_klubu;
```

Zapis ten wydaje się być bardziej uniwersalnym w kontekście innych, niżej opisanych konstrukcji, ale dla równozłączenia nie ma żadnej różnicy.

Składnia złączenia wewnętrznego dla wielu tabel:

W przypadku większej liczby tabel należy zarówno ich nazwy, jak i nazwy kolumn umieścić w odpowiednich frazach zdania `select`, jak również rozbudować odpowiednio warunki `where` lub `join`, Np. poniższe zdania:

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii
      from bd3_zawodnicy, bd3_kluby, bd3_kategorie
            where bd3_zawodnicy.nr_klubu = bd3_kluby.nr_klubu
                  and bd3_zawodnicy.nr_kategorii = bd3_kategorie.nr_kategorii;
```

lub równoważne jemu:

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii
      from bd3_zawodnicy
            join bd3_kluby on bd3_zawodnicy.nr_klubu = bd3_kluby.nr_klubu
            join bd3_kategorie on bd3_zawodnicy.nr_kategorii = bd3_kategorie.nr_kategorii;
```

dadzą wynik:

IMIE	NAZWISKO	NAZWA_KLUBU	NAZWA_KATEGORII
Piotr	Kuczkowski	KB Lotos Jabłonna	IV
Aleksander	Cieślak	Warszawianka	IV
Tadeusz	Strzałkowski	KB Trucht Warszawa	V
Jan	Kowalski	Allianz Warszawa	IV
Grzegorz	Grabowski	KB Trucht Warszawa	III
Petter	Cieślak	KB Gymnasium Warszawa	III
Leszek	Kowalczyk	KB Pułaski Strong Warka	VII
Marcin	Grzana	Allianz Warszawa	VI
Rafał	Radomski	Allianz Warszawa	II
Wojciech	Gaca	KB Gymnasium Warszawa	V
Tomasz	Jakubowski	KB Trucht Warszawa	III

.....

W takich konstrukcjach zdania `select` można stosować aliasy nazw tabel w celu skrócenia zapisu.

Na przykład:

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii
from bd3_zawodnicy z , bd3_kluby kl , bd3_kategorie ka
where z.nr_klubu = kl.nr_klubu
and z.nr_kategorii = ka.nr_kategorii;
```

lub

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii
from bd3_zawodnicy z
join bd3_kluby kl on z.nr_klubu = kl.nr_klubu
join bd3_kategorie ka on z.nr_kategorii = ka.nr_kategorii;
```

W przypadku zdefiniowania takich aliasów, muszą one być używane w całym zdaniu do specyfikowania tabel, a w szczególności we frazach *where* lub *join* do definiowania złączeń. Niedopuszczalne jest w takim przypadku używanie w jednym zdaniu *select* zarówno pełnej nazwy tabeli, jak i jej aliasu.

W równozłączniach można rozbudowywać frazę *where* o różnego rodzaju filtry według zasad obowiązujących dla pobierania danych z jednej tabeli, jak również sortować zbiór wynikowy.

Na przykład zdanie:

```
select nazwisko || ' ' || imie as Zawodnik, nazwa_klubu as Klub
,data_urodzenia as "Data urodzenia"
from bd3_zawodnicy z , bd3_kluby kl
where z.nr_klubu = kl.nr_klubu
and data_urodzenia between '87/01/01' and '87/12/31'
and plec= 'K';
order by "Data urodzenia";
```

lub równoważne mu:

```
select nazwisko || ' ' || imie as Zawodnik, nazwa_klubu as Klub
,data_urodzenia as "Data urodzenia"
from bd3_zawodnicy z join bd3_kluby kl on z.nr_klubu = kl.nr_klubu
where data_urodzenia between '87/01/01' and '87/12/31'
and plec= 'K'
order by "Data urodzenia";
```

utworzy zbiór wynikowy:

Zawodnik	Klub	Data urodzenia
Dobkowska Justyna	KB Lechici Zielonka	87/02/19
Butkiewicz Anna	KB Gymnasium Warszawa	87/02/23
Sawicka Jolanta	KB Trucht Warszawa	87/09/11

Gdyby w powyższym przykładzie zaszła potrzeba, aby zbiór wynikowy zawierał kolumnę *nr_kategorii* – to należy zauważyć, że ta kolumna występuje w dwóch tabelach: BD3_KATEGORIE i BD3_ZAWODNICY. Próba wykonania zdania:

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii, nr_kategorii
from bd3_zawodnicy z , bd3_kluby kl , bd3_kategorie ka
where z.nr_klubu = kl.nr_klubu and z.nr_kategorii = ka.nr_kategorii;
```

zakończy się niepowodzeniem:

```
| Error report -
| SQL Error: ORA-00918: kolumna zdefiniowana w sposób niejednoznaczny
| 00918. 00000 - "column ambiguously defined"
```

gdyż nie zostało określone, z której tabeli mają być pobierane dane dotyczące nr_kategorii.

Należy zmodyfikować zdanie poprzez dokładne wyspecyfikowanie tej kolumny nazwą tabeli lub jej aliasem:

```
select imie, nazwisko, nazwa_klubu, nazwa_kategorii, z.nr_kategorii
from bd3_zawodnicy z , bd3_kluby kl , bd3_kategorie ka
where z.nr_klubu = kl.nr_klubu and z.nr_kategorii = ka.nr_kategorii;
```

W tym przypadku kolumna nr_kategorii będzie przyjmowała wartości z tabeli BD3_ZAWODNICY.

Złączenie zewnętrzne

Złączenie zewnętrzne (*outer join*) polega na dopisaniu do zbioru wynikowego wszystkich wierszy z tabel źródłowych, niezależnie czy istnieją odpowiadające im wiersze w tych tabelach. Jeśli nie można znaleźć odpowiednika danego wiersza w innej tabeli, pola te w zbiorze wynikowym przyjmują wartość *null*.

Istnieją trzy rodzaje złączeń zewnętrznych:

Złączenie zewnętrzne lewostronne:

Złączenie to (*left outer join* lub *left join*) lewostronnie umieszcza w zbiorze wynikowym wszystkie wiersze z pierwszej (lewej) z podanych tabel:

tabela1 *left outer join* tabela2

Poniżej dla porównania pokazano to samo zdanie SQL w dwóch sposobach łączenia tabel.

Pierwszy sposób – złączenie wewnętrzne (*join*) omówione wcześniej:

```
select nazwa_klubu, nazwisko
      from bd3_kluby kl
            join bd3_zawodnicy z on z.nr_klubu = kl.nr_klubu
      order by kl.nr_klubu desc;
```

Należy zwrócić uwagę na specyfikację kolumny, według której następuje sortowanie zbioru wynikowego. Kolumna nr_klubu występuje w obu łączonych tabelach i dlatego trzeba precyzyjnie określić, według której ma nastąpić porządkowanie zbioru. Dodatkowo kolumna ta nie występuje w zbiorze wyników (brak jej we frazie *select*)³.

NAZWA_KLUBU	NAZWISKO
Skra Warszawa	Zatorski
KB Lechici Zielonka	Katłubaj-Domżak
KB Lechici Zielonka	Dobkowska
KB Lechici Zielonka	Karczewski
KB Lechici Zielonka	Popowski

...

³ Gdyby na liście frazy *select* znajdowała się, odpowiednio wyspecyfikowana, kolumna nr_klubu, wtedy fraza *order by* może zawierać nazwę tej kolumny bez specyfikacji.

Zbiór wyników zawiera zestawienie zawodników, którzy przynależą do jakiegoś klubu czyli w zbiorze wyników nie znajdują się zawodnicy, dla których w tabeli BD3_ZAWODNICY w kolumnie nr_klubu występuje *null*. Liczność tego zbioru wynosi 766, mimo, że w ewidencji jest ich 771.

Drugi sposób – złączenie zewnętrzne lewostronne (*left outer join*):

```
select nazwa_klubu, nazwisko
from bd3_kluby kl
left outer join bd3_zawodnicy z on z.nr_klubu = kl.nr_klubu
order by nazwisko nulls first;
```

da następujący wynik:

NAZWA_KLUBU	NAZWISKO
1 ACTION Warszawa	(null)
2 KU AZS WAT Warszawa	(null)
3 Grunwald Poznań	(null)
4 KB Galeria Warszawa	(null)
5 Flota Gdynia	(null)
6 Śląsk Wrocław	(null)
7 Allianz Warszawa	Adach
8 KB Lechici Zielonka	Adamczyk
.....	

Wyraźnie widać, że w zbiorze wynikowym pojawiło się sześć wierszy, których nie było w przypadku łączenia wewnętrznego. Dla tych wierszy kolumny pochodzące z tabeli prawej są uzupełnione wartościami *null*.

Zestawienie pokazuje, że istnieją w bazie kluby, który nie posiadają żadnego zawodnika. Liczność tego zbioru wynikowego wynosi 772 (766 zawodników przynależnych do jakiegoś klubu i dodatkowo sześć klubów bez zawodników).

Złączenie zewnętrzne prawostronne:

Złączenie to (*right outer join* lub *right join*) prawostronne umieszcza w zbiorze wynikowym wszystkie wiersze z drugiej (prawej) z podanych tabel:

tabela1 *right outer join* tabela2

np. wyżej podany przykład złączenia lewostronnego, w którym zmieniono klauzulę *left* na *right* wygląda jak poniżej:

```
select nazwa_klubu, nazwisko
from bd3_kluby kl
right outer join bd3_zawodnicy z on z.nr_klubu = kl.nr_klubu
order by kl.nr_klubu desc;
```

i da efekt:

NAZWA_KLUBU	NAZWISKO
(null)	Misiejuk
(null)	Kostecki
(null)	Wereda
(null)	Małkiewicz
(null)	Wojciuk
Skra Warszawa	Zatorski
KB Lechici Zielonka	Misiaczek
KB Lechici Zielonka	Młotkowski
.....	

czyli w zbiorze wynikowym znajdą się wszystkie wiersze tabeli BD3_ZAWODNICY (771) uzupełnione o wartości z tabeli BD3_KLUBY, a w przypadku braku odpowiadających wierszy tej tabeli w kolumnie nazwa_klubu pojawią się wartości *null*.

Warto zaznaczyć, że konstrukcja:

tabela1 left outer join tabela2

jest równoważna konstrukcji:

tabela2 right outer join tabela1

Złączenie zewnętrzne pełne:

Złączenie zewnętrzne pełne (*full outer join* lub *full join*) uwzględnia wszystkie wiersze z obu tabel składowych, wypełniając odpowiednie kolumny wartościami *null* tam, gdzie to konieczne.

tabela1 full outer join tabela2

Jeśli tabele składowe łączy związek między kluczem głównym a kluczem obcym, wówczas rezultat tego złączenia będzie pokrywał się z wynikami przeprowadzenia złączenia lewostronnego, w którym tabela zawierająca klucz główny znajduje się po lewej stronie operatora *join* oraz złączenia prawostronnego, w którym tabela ta leży po prawej stronie operatora.

Innymi słowy do zbioru wynikowego przejdą wszystkie wiersze z lewej tabeli oraz wszystkie wiersze z prawej tabeli. Pola wierszy, które nie mogły być połączone będą miały wartość *null*.

Zdanie:

```
select nazwa_klubu, nazwisko
from bd3_kluby kl
full outer join bd3_zawodnicy z on z.nr_klubu = kl.nr_klubu
order by nazwa_klubu nulls first;
```

utworzy zbiór:

NAZWA_KLUBU	NAZWISKO
(null)	Misiejuk
(null)	Kostecki
(null)	Wereda
(null)	Małkiewicz
(null)	Wojciuk
ACTION Warszawa	(null)
Akvedukt Kielce	Walczak
Akvedukt Kielce	Augustyniak

...

Liczność tego zbioru wynosi 777, gdyż jest 771 zawodników w ewidencji i dodatkowo sześć klubów bez zawodników.

Uwaga:

Jeśli w bazie danych relacje między dwiema tabelami (klucz główny i klucz obcy) są tak zdefiniowane, że dopuszcza się wartość *null* na pozycji klucza obcego to należy być tego świadomym konstruując zapytania *select*, aby nie utracić potencjalnie ważnych danych. Taki przypadek zachodzi w implementacji omawianego modelu między tabelami dotyczącymi ewidencji zawodników i ewidencji klubów. Natomiast nie zachodzi w relacji zawodnicy i kategorie wiekowe, gdyż w definicji tabeli

BD3_ZAWODNICY kolumna nr_kategorii nie może przyjmować wartości *null*, co oznacza, że każdy zawodnik musi mieć określoną kategorię wiekową.

Operacja unii

Operacja unii (*union*) tworzy zbiór wynikowy na podstawie dwóch lub większej liczby zdań *select*. W przeciwnieństwie dołączenia, które zwiększa liczbę kolumn w zbiorze wynikowym, operacja unii zmienia liczbę wierszy.

Np.:

```
select nazwisko || ' ' || imie as "Nazwisko i imię", rezultat_min, rezultat_sek, nazwa_klubu, plec
from bd3_zawodnicy z, bd3_wyniki w, bd3_kluby k
where z.nr_klubu = k.nr_klubu and z.nr_zawodnika = w.nr_zawodnika
      and w.nr_zawodow = 1 and plec = 'M' and k.nr_klubu = 4
```

union

```
select nazwisko || ' ' || imie as "Nazwisko i imię", rezultat_min, rezultat_sek, nazwa_klubu, plec
from bd3_zawodnicy z, bd3_wyniki w, bd3_kluby k
where z.nr_klubu = k.nr_klubu and z.nr_zawodnika = w.nr_zawodnika
      and w.nr_zawodow = 1 and plec = 'K' and k.nr_klubu = 20
```

```
order by 5 desc, 2,3;
```

Powyższa konstrukcja składa się z czterech części.

Pierwsze zdanie *select* tworzy zbiór wynikowy zawierający wyspecyfikowane kolumny w celu pokazania rezultatów zawodników w zawodach o numerze 1 będących mężczyznami należącymi do klubu o identyfikatorze 4. Pozostałe warunki ograniczające we frazie *where* służą do zdefiniowania odpowiedniego połączenia tabel.

Drugim elementem jest fraza *union* lub *union all*.

W dalszej części występuje drugie zdanie *select* tworzące zbiór wynikowy zawierający te same kolumny jak w zdaniu pierwszym, ale z innymi ograniczeniami. Tym razem wybrane zostaną kobiety biorące udział w zawodach o numerze 1, które należą do klubu o numerze 20.

Końcowym elementem, który może (ale nie musi) wystąpić jest sposób sortowania. W przykładzie zastosowano skróconą formę zapisu frazy *order by* zamieniając nazwy kolumn na numery określające miejsce ich występowania w zdaniu *select*.

Nazwisko i imię	REZULTAT_MIN	REZULTAT_SEK	NAZWA_KLUBU	PLEC
Ławecki Tomasz	37	22	Legia Warszawa	M
Harkot Marta	59	48	KB Lotos Jabłonna	K
Gołabek Agnieszka	63	5	KB Lotos Jabłonna	K

Uwagi:

1. W przypadku, gdy zbiory wynikowe obu zdań *select* mają powtarzające się wiersze (co nie zachodzi w powyższym przykładzie, gdyż mamy do czynienia z dwoma rozłącznymi zbiorami z uwagi na płeć) fraza *union* automatycznie ją wyeliminuje czyli w tle zostanie zastosowana klauzula *distinct*. Jeżeli chcemy, aby w końcowym zbiorze wynikowym występowły powtarzające się wiersze należy zastosować frazę *union all*.
2. W obu zdaniach *select* liczba kolumn musi być taka sama, jak również musi zachodzić zgodność typów lub przynajmniej możliwość konwersji na jeden rodzaj typu.
3. Frazy *order by* można użyć tylko raz na końcu unii i nazwy kolumn w niej zawarte muszą występować na liście kolumn pierwszego zdania *select*.
4. Unia może zawierać większą liczbę zdań *select* połączonych frazą *union* lub *union all*.

Zadania do samodzielnego wykonania:

Poniższe zadania wykonać przy następujących założeniach:

- We wszystkich zadaniach pobierać dane z minimum trzech tabel,
- Kolumny Imię i Nazwisko łączyć w jedną kolumnę,
- Stosować aliasy do zmiany nazw kolumn,
- Zestawienia powinny zawierać takie kolumny jak Nazwisko i imię, Nazwa klubu, Nazwa kategorii.

1. Opracować raport zawierający zestawienie mężczyzn, którzy brali udział w zawodach numer 1 i 3 i uzyskali wynik między 44 i 48 min (kolumny Rezultat_Min i Rezultat_Sek). Posortować zbiór według numeru zawodów i osiągniętego czasu.
2. Opracować zestawienie kobiet, które należą do klubów o numerach od 5 do 40, urodzone są w latach między 1975 i 1984 i które nie zdobyły punktów w klasyfikacji generalnej (kolumna Punkty_Globalne).
3. Opracować komunikat końcowy zawodów numer 3 pokazujący klasyfikację zawodników (mężczyzn) na mecie według osiągniętych czasów. Wykorzystać kolumnę Lokata_W_Biegu.
4. Przy pomocy unii zaprojektować raport pokazujący zestawienie mężczyzn należących do kategorii IV i kobiet urodzonych w latach 1970 – 1985, których nazwiska zaczynają się na literę K i kończą na 'ska'. Posortować zbiór według roku urodzenia i nazwiska.
5. Wykonać zadanie z pkt. 4 korzystając z jednego zdania *select*.
6. Dokonać modyfikacji modelu bazy danych. O każdym klubie należy przechowywać informacje teleadresowe, rok założenia klubu, nazwisko prezesa klubu itp. Dane te należy zapisywać w osobnej tabeli. Dokonać odpowiedniej modyfikacji relacji między tabelami poprzez napisanie skryptu realizującego te modyfikacje. Wprowadzić przykładowe dane dla kilku klubów i napisać zdanie wybierające informacje z bazy przy uwzględnieniu nowej tabeli.
7. Dokonać modyfikacji modelu bazy danych. O każdym zawodniku trzeba przechowywać informacje o jego osiągnięciach w postaci wpisów do osobnej tabeli (np. rok i zdarzenie) przy założeniu, że jeden zawodnik może mieć kilka wpisów. Wpisy dotyczą historii zawodnika (np. zmiana barw klubowych, wyniki sportowe). Dokonać odpowiedniej modyfikacji relacji między tabelami poprzez napisanie skryptu realizującego te modyfikacje. Wprowadzić przykładowe dane dla kilku zawodników i napisać zdanie wybierające informacje z bazy przy uwzględnieniu nowej tabeli.
8. Dokonać modyfikacji modelu bazy danych tak, aby niemożliwe było zarejestrowanie zawodnika bez przynależności klubowej. Utworzyć fikcyjny klub o nazwie Niezrzeszeni i w nim umieścić dotychczasowych zawodników bez przydziału.
9. Dokonać modyfikacji modelu bazy danych polegającej na konieczności automatycznego rejestracji osoby i czasu wprowadzenia danych o zawodniku do ewidencji oraz ewentualnej ich modyfikacji. Należy wykorzystać systemowe funkcje USER i SYSDATE.
10. W procesie wczytywania danych do tabel z plików płaskich przy pomocy SQL Developer można było utworzyć skrypt zawierający kompletne zdania *insert* wybierając metodę *Insert Script*. Skrypty te były tworzone automatycznie w odpowiednim folderze systemu operacyjnego (c:\users\%user%\AppData\Local\Temp). Należy, na ich podstawie utworzyć jeden zbiorczy skrypt zawierający wszystkie zdania *insert* niezbędne do wprowadzenia danych do wszystkich tabel, zakończyć go zatwierdzeniem transakcji oraz zdaniem sprawdzającym zawartość tabel. Nazwać tak powstały skrypt *lab_BD3_populate.sql*, a następnie dokonać globalnego testowania poprawności wdrożenia przy pomocy sekwencji poleceń:
@c:\temp\lab_BD3_drop.sql
@c:\temp\lab_BD3_create.sql
@c:\temp\lab_BD3_populate.sql

Bazy Danych laboratorium

**Laboratorium
BD4**

Na potrzeby zajęć zostanie wykorzystany model bazy danych opisany i zaimplementowany w ramach Laboratorium BD3.

Wykonywanie obliczeń w zdaniach SQL

W zdaniach SQL można wykonywać proste operacje arytmetyczne na wartościach kolumn i na stałych używając do tego celu klasycznych operatorów działań: +, -, *, / oraz ().

Przykładowo:

Zdanie:

```
select 125 + 15 / 50 - 3 as "I przykład",
       (125 + 15) / 50 - 3 as "II przykład",
       (125 + 15) / (50 - 3) as "III przykład"
  from dual;
```

da poniższe wyniki:

I przykład	II przykład	III przykład
122,3	-0,2 2,97872340425531914893617021276595744681	

Należy zwrócić uwagę na konstrukcję zdania `select`. W podstawowej swej formie jego definicja zawiera frazę `from: select from tabela`. Implementacja SQL w środowisku Oracle nie dopuszcza skróconej formy `select` bez frazy `from`, w przypadku kiedy dane lub obliczenia nie pochodzą z żadnej tabeli¹.

Jedną z wielu funkcji operujących na danych liczbowych jest funkcja zaokrąglenia, której ogólna postać jest następująca:

`round (zmienna liczbową, n),`

gdzie : zmienna liczbową może być nazwą kolumny tabeli lub wartością,
n – dokładność zaokrąglenia.

Przykładowo powyższe zdanie SQL z tą funkcją (domyślnie n=0):

```
select round (125 + 15 / 50 - 3) as "I przykład",
       round ((125 + 15) / 50 - 3) as "II przykład",
       round ((125 + 15) / (50 - 3)) as "III przykład"
  from dual;
```

da wynik:

I przykład	II przykład	III przykład
122	0	3

¹ W innych systemach zarządzania bazami danych (np. Sybase) nie jest to konieczne i frazę `from` można pominąć. Aby być w zgodzie z klasyczną formą zdania `select` wprowadza się fikcyjną tabelę (`dual` dla Oracle, `dummy` dla Sybase), której można używać w formie: `select ... from dual`.

Zdanie:

```
select round (25.123456 , 2) as "Round to 2",
       round (25.123456 , 4) as "Round to 4",
       round (25.123456 , 0) as "Round to 0",
       round (25.123456 , -1) as "Round to -1"
  from dual;
```

da poniższe wyniki:

Round to 2	Round to 4	Round to 0	Round to -1
25,12	25,1235	25	30

Oczywiście obliczeń można dokonywać także na danych pochodzących z tabel. Przykładowo poniższe zdanie generuje zbiór wynikowy, w którym obliczono wiek zawodników:

```
select imie ||' '| nazwisko as "Zawodnik", nazwa_klubu as "Klub",
       round (sysdate - data_urodzenia) as "Wiek"
  from bd3_zawodnicy z, bd3_kluby k
 where z.nr_klubu = k.nr_klubu and z.nr_klubu = 3
 order by "Wiek" desc;
```

Zawodnik	Klub	Wiek
Edmund Werthein	KB Gymnasion Warszawa	33807
Bożena Gradus	KB Gymnasion Warszawa	33163
Wojciech Bielowicki	KB Gymnasion Warszawa	31844
Alek Borowski	KB Orientuz Warszawa	31226
Adam Gorzkowski	KB Lechici Zielonka	31156
Jan Maliszewski	Bielanski KB Warszawa	31115
Jarosław Żelazko	KB Gymnasion Warszawa	31078

Powyższy przykład ma tę wadę, że wiek obliczony jest w dniach po odjęciu dwóch dat od siebie. Nic nie stoi na przeszkodzie, aby prostym działaniem arytmetycznym przeliczyć dni na lata. Natomiast zaletą jest użycie zmiennej `sysdate`, która jako zmienna globalna zawiera bieżącą datę na serwerze bazodanowym.

Wykonywanie operacji na łańcuchach znakowych

Istnieje szereg funkcji pozwalających na przetwarzanie łańcuchów tekstowych. Jedną z nich jest omówiony w Laboratorium BD2 operator konkatenacji (łączenia łańcuchów), którego zapis wygląda tak:

```
kolumna_1 || kolumna_2 || '...łańcuch znaków...' || .....
```

Podobną rolę pełni funkcja `concat`, której ogólna postać wygląda tak:

```
concat(ciąg_znaków_1, ciąg_znaków_2)
```

Wadą tej funkcji jest to, że jest dwuargumentowa. Chcąc jej użyć do łączenia imienia z nazwiskiem należy zastosować zagnieżdżenie funkcji:

```
..... concat( concat(imie, ' '), nazwisko ) .....
```

Najbardziej popularnymi funkcjami przetwarzającymi łańcuchy znakowe są funkcje:

- *upper*,
- *lower*,
- *substr*,
- *initcap*.

Funkcje *upper* i *lower*

Funkcje *upper* i *lower* są funkcjami zamieniającymi cały łańcuch znaków odpowiednio na duże i małe litery:

```
select lower ( imie ) as "Imię", upper ( nazwisko ) as "Nazwisko"
from bd3_zawodnicy;
```

Imię	Nazwisko
piotr	KUCZKOWSKI
aleksander	CIEŚLAK
tadeusz	STRZAŁKOWSKI
jan	KOWALSKI
grzegorz	GRABOWSKI
petter	CIEŚLAK

Użycie tych funkcji ma dwa praktyczne zastosowania.

Pierwsze to wygląd opracowywanych raportów. Na powyższym przykładzie widać, że końcowy raport będzie zawierał nazwiska zapisane dużymi literami, chociaż w bazie tak nie jest.

Drugie zastosowanie dotyczy metod przeszukiwania bazy i jej modyfikacji.² Jeśli baza danych w momencie tworzenia zostanie ustawiona na rozróżnianie wielkości liter w danych, to nie bez znaczenia jest czy warunek filtrowania będzie napisany tak:

```
.....  
where imie = 'stefan'
```

czy też tak:

```
.....  
where imie = 'Stefan'
```

W takim przypadku, aby ustrzec się przed błędami wyszukiwania bezpieczne jest użyć jednej z tych funkcji w warunku wyboru, np.:

```
select lower ( imie ) as "Imię", upper ( nazwisko ) as "Nazwisko"
from bd3_zawodnicy z, bd3_kluby k
where z.nr_klubu = k.nr_klubu and upper ( imie ) = 'STEFAN';
```

Funkcja *upper* we frazie *where* powoduje, że wszystkie imiona z tabeli BD3_ZAWODNICY są zamieniane na duże litery i porównywane z napisem 'STEFAN'. W przypadku zgodności wybrane wiersze przechodzą do zbioru wyników, w którym imiona są zamieniane na małe litery zgodnie z listą we frazie *select*.

To rozwiązanie jest stosowane bardzo często w aplikacjach bazodanowych, w których dane pochodzą z formularzy wypełnianych przez użytkownika. Aby zachować jednolitość bazy można w zdaniach *insert*, *update* stosować przekształcenia łańcuchów znaków.

² W bazie oraclowej na serwerze *city.wsisiz.edu.pl* ten przypadek zachodzi czyli 'abacki' nie oznacza tego samego co 'Abacki' lub 'ABACKI' (*case sensitivity*). Można spotkać inne instalacje, w których ta zależność nie zachodzi.

Np.

```
insert into tabela (.....)
values
(
    upper( :zm_1 ),
    upper( :zm_2 ),
    .....
);
lub:
```

```
update tabela
set kolumna_N = lower( :zm_N )
where kolumna_M = lower( :zm_M );
```

Uwaga:

Nazwy zmiennych poprzedzone znakiem dwukropka oznaczają zmienne wiązania (*bind variable*), które służą do przekazywania danych między środowiskami programistycznymi, takimi jak java, php, c#, html i środowiskiem bazodanowym czyli np. SQL czy PL/SQL. Typowym przykładem są formularze aplikacji, z których po ich wypełnieniu wartość pól łączona jest ze zdaniem SQL (*parsowanie*) w celu wprowadzenia danych do tabeli lub przeszukiwania bazy.

Funkcja **substr**

Funkcja *substr* wycina fragment z podanego łańcucha znaków. Ogólna postać tej funkcji wygląda tak:

```
substr (łańcuch znaków, pozycja startowa, liczba znaków)
```

Przykładowo chcąc utworzyć trzyliterowe skróty nazwisk zawodników można użyć konstrukcji:

```
select nazwisko as " Nazwisko", substr (nazwisko, 1, 3) as "Identyfikator"
from bd3_zawodnicy;
```

która da poniższy zbiór wyników:

Nazwisko	Identyfikator
Kuczkowski	Kuc
Cieślak	Cie
Strzałkowski	Str
Kowalski	Kow
Grabowski	Gra
Cieślak	Cie

Funkcja **initcap**

Funkcja *initcap* zamienia podany argument w ten sposób, że każdy pierwszy znak słowa zamienia na dużą literę, a pozostałe czyni małymi. Ogólna postać wygląda tak:

```
initcap ( ciąg_znaków )
```

Przykładowo zdanie:

```
select initcap ( 'ADAM ABACKI' ) as "Nazwisko" from dual;
```

da poniższy efekt:

 Nazwisko
Adam Abacki

Uwaga:

Istnieje szereg innych funkcji wzbogacających specyfikację języka SQL i PL/SQL. Szczegółowy opis można znaleźć w dokumentacji Oracle (lub innych serwerów bazodanowych). Dodatkowo warto zapoznać się z zawartością serwisów [Tech on the Net](#) lub [Oracle Tutorials](#).

Wykonywanie operacji na datach

Data w systemach bazodanowych przechowywana jest jako wartość liczbową mieszana. Część całkowita określa liczbę dni jaką upłynęła od pewnego dalekiego w przeszłości dnia (np. dla Oracle jest to dzień 1 stycznia 4712 p.n.e.), a część ułamkowa, jeśli występuje, wskazuje na czas w ramach jednej doby (np. wartość 0,5 określa południe) z dokładnością do milisekund.

Drugim ważnym aspektem związanym z datami jest sposób wprowadzania dat do bazy danych.

Załóżmy, że chcemy wprowadzić do tabeli BD3_ZAWODY informację o kolejnych zawodach:

```
insert into bd3_zawody ( data_zawodow, nazwa_zawodow, nr_zawodow )
values ( '24/02/2018', 'Grand Prix Warszawy', 5);
```

W momencie uruchomienia tego zdania pojawi się błąd:³

```
Error starting at line : 1 in command -
insert into bd3_zawody (data_zawodow, nazwa_zawodow, nr_zawodow)
values ('24/02/2018', 'Grand Prix Warszawy', 5)
Error report -
ORA-01830: wzorzec formatu daty kończy się przed konwersją całego napisu wejściowego
```

wskazujący na zły format wprowadzanej daty.

Jak zatem postępować chcąc wprowadzać dane typu data do bazy danych?

Po pierwsze trzeba się dowiedzieć, jaki format daty jest przyjęty za standardowy na konkretnym serwerze. Format ten ustala administrator serwera w momencie jego instalowania. Aby to zrobić należy odczytać z serwera datę systemową.

Poniżej zostanie zaprezentowane zdanie pokazujące użycie dwóch funkcji standardowych określających zmienne systemowe przechowujące datę i czas systemowy.

```
select sysdate    as "Data systemowa",
       systimestamp as "Czas systemowy dokładny"
  from dual;
```

 Data systemowa	 Czas systemowy dokładny
18/09/27	18/09/27 15:28:51,463109000 +02:00

³ Należy zwrócić uwagę, że daty, tak samo jak wartości znakowe, opatrzone są apostrofami jako znakami ograniczającymi.

Po drugie do wprowadzania dat używać odczytanego formatu. W tym przypadku jest to format określany jako 'YYYY/MM/DD' mimo, że na powyższym rysunku widnieje 'YY/MM/DD'. Zostanie to wyjaśnione w dalszej części materiału.

Zatem zdanie wprowadzające nową pozycję do tabeli BD3_ZAWODY powinno wyglądać tak:

```
insert into bd3_zawody ( data_zawodow, nazwa_zawodow, nr_zawodow )
values ( '2018/02/24', 'Grand Prix Warszawy', 5);
```

Na datach można wykonywać operacje arytmetyczne, ale w ograniczonym zakresie.

Typowymi i najczęściej spotykanymi działańami są:

- | | |
|----------------|-----------------------------------------------|
| date + integer | - dodanie do daty pewnej liczby dni, |
| date - integer | - odjęcie od daty pewnej liczby dni, |
| date - date | - obliczenie liczby dni między dwiema datami, |

Przykładowo zdanie:

```
select sysdate      as "Data systemowa",
       sysdate + 5   as "Termin końcowy"
  from dual;
```

daje wynik:

Data systemowa	Termin końcowy
18/09/27	18/10/02

ale wykonanie zdania:

```
select '2018/09/26' + 5 from dual;
```

jest niemożliwe:

```
Error starting at line : 1 in command -
select '2018/09/26' + 5 from dual
Error report -
ORA-01722: niepoprawna liczba
```

Dlaczego?

Zapis '2018/09/26' nie jest traktowany jako data, jeśli nie jest odczytywany z tabeli. W powyższym przykładzie jest on traktowany jako ciąg znaków i nie można go używać do działań arytmetycznych.

Natomiast możliwe jest używanie tego zapisu w zdaniach *insert*, *update*, *delete* i we frazie *where*, np.:

```
.....  
where data_urodzenia between '1986/01/01' and '1986/12/31'  
.....
```

Chcąc użyć daty w takiej postaci do bezpośrednich obliczeń należy najpierw ją w sposób jawnego przekonwertować czyli użyć funkcji konwersji.

Funkcje konwersji

W celu jawnnej zmiany typu przetwarzanych danych stosuje się funkcje konwersji. Standard Oracle przewiduje między innymi takie funkcje konwersji: `to_date`, `to_char`, `to_number`, `to_timestamp` itp. Niektóre z nich zostaną omówione poniżej.

`to_date`:

Podstawową funkcją konwersji jest funkcja `to_date`, która zamienia datę w postaci napisu na datę w postaci wyrażenia. Ogólna postać tej funkcji wygląda tak, jak poniżej:

```
to_date ( napis_będący_data , format )
```

gdzie:

napis_będący_data - ciąg znaków, który ma być przekonwertowany na datę,
 format - informacja dla interpretera, w jakiej postaci jest przedstawiony napis_będący_data,
 jest on budowany z znaków symbolizujących lata (YYYY), miesiące (MM lub MON lub MONTH) i dni (DD).

Na przykład zdanie:

```
select to_date ( '2018.09.26', 'YYYY.MM.DD' ) + 5 as "Termin I",
       to_date ( '26-WRZ-2018', 'DD-MON-YYYY' ) + 10 as "Termin II",
       to_date ( '26-WRZESIEN-2018', 'DD-MONTH-YYYY' ) + 15 as "Koniec"
  from dual;
```

daje wynik:

Termin I	Termin II	Koniec
18/10/01	18/10/06	18/10/11

Uwaga:

Mimo, że nastąpiła konwersja daty do postaci YYYY/MM/DD, sposób jej prezentacji wygląda jako YY/MM/DD. Aby dociec dlaczego, należy odczytać parametry NLS (*National Language Support*) serwera bazodanowego. Są to parametry określające między innymi język narodowy, symbol waluty narodowej i formaty daty i czasu ustawione dla danego serwera. Wykonać to można zdaniem:

```
select parameter, value from v$nlsparameters;
```

Widać, że językiem jest polski czyli polskie znaki będą prawidłowo interpretowane, a data i czas mają format RR/MM/DD. Format RR określa rok w postaci dwuznakowej i różni się od YY.

Aby się o tym przekonać można do tabeli BD3_ZAWODY wprowadzić dane przy pomocy poniższych zdań:

```
insert into bd3_zawody ( data_zawodow, nazwa_zawodow, nr_zawodow )
  values (to_date ( '98-02-24', 'YY-MM-DD' ), 'Grand Prix Warszawy', 5);
insert into bd3_zawody ( data_zawodow, nazwa_zawodow, nr_zawodow )
  values (to_date ( '98-02-24', 'RR-MM-DD' ), 'Grand Prix Warszawy', 6);
```

Po odpytaniu tabeli zdaniem `select * from bd3_zawody` otrzymamy wyniki:

NR_ZAWODOW	NAZWA_ZAWODOW	DATA_ZAWODOW	PODTYTUL
1	Cztery Pory Roku - Jesień	10/10/22	(null)
2	Cztery Pory Roku - Zima	11/02/19	(null)
3	Cztery Pory Roku - Wiosna	11/05/21	(null)
4	Cztery Pory Roku - Lato	11/09/03	im. S. Dankowskiego
5	Grand Prix Warszawy	98/02/24	(null)
6	Grand Prix Warszawy	98/02/24	(null)

Pozornie obie daty w nowowprowadzonych wierszach wyglądają tak samo. Ale jest różnica.

Każdy użytkownik może chwilowo zmienić parametry NLS w ramach swojej bieżącej sesji. Należy użyć na przykład takiego zdania:

```
alter session
set nls_date_format = 'YYYY/MM/DD';
```

Po jego wykonaniu ponowne odpytanie tabeli BD3_ZAWODY da inny wynik:

NR_ZAWODOW	NAZWA_ZAWODOW	DATA_ZAWODOW	PODTYTUŁ
1	Cztery Pory Roku - Jesień	2010/10/22	(null)
2	Cztery Pory Roku - Zima	2011/02/19	(null)
3	Cztery Pory Roku - Wiosna	2011/05/21	(null)
4	Cztery Pory Roku - Lato	2011/09/03	im.S.Dankowskiego
5	Grand Prix Warszawy	2098/02/24	(null)
6	Grand Prix Warszawy	1998/02/24	(null)

W przypadku, gdy do tabeli wprowadzana jest data, w której rok jest określony na dwóch pozycjach to:

- jeśli jest to format YY i rok >= 50 to jest on umieszczany w XXI wieku,
- jeśli jest to format RR i rok >= 50 to jest on umieszczany w XX wieku,
- jeśli rok < 50 to bez względu na format (YY lub RR) jest on umieszczany w XXI wieku.

Wnioski końcowe:

- Bardzo często administratorzy baz danych mają do czynienia z sytuacją, że muszą wczytywać do bazy danych (hurtowni) dane z plików pochodzących z różnych systemów (na przykład w korporacjach globalnych). Formaty dat mogą się różnić między źródłami danych i serwerami bazodanowymi. Powyższe rozważania mogą te czynności znakomicie przyspieszyć, gdyż łatwo sobie wyobrazić jakim nakładem opatrzoną jest modyfikacja pliku płaskiego zawierającego, na przykład, jeden milion wierszy z datą niezgodną z docelowym formatem.
- Ustawienie parametrów NLS przy pomocy konstrukcji *alter session ...* jest obowiązujące do czasu zamknięcia sesji. Po ponownym zalogowaniu się do schematu obowiązują parametry ustawione na serwerze, a nie w sesji.

extract:

Jest to funkcja dokonująca ekstrakcji jednego z elementów daty lub czasu, takich jak rok, miesiąc, dzień, godzina, minuta czy sekunda. Nazwy atrybutów tej funkcji są synonimami ekstrahowanych wielkości (w języku angielskim):

Zdanie

```
select extract ( year from sysdate ) as Rok
      , extract ( month from sysdate ) as Miesiąc
      , extract ( day from sysdate ) as Dzień
from dual;
```

zwraca wyniki:

ROK	MIESIĄC	DZIEŃ
2018	9	28

A zdanie:

```
select extract ( hour from systimestamp ) as Godzina
      , extract ( minute from systimestamp ) as Minuta
      , extract ( second from systimestamp ) as Sekunda
from dual;
```

zwraca wyniki:

GODZINA	MINUTA	SEKUNDA
18	47	38,928741

Oczywiście zamiast zmiennej globalnej `sysdate` czy `systimestamp` można użyć dowolnej innej wartości będącej datą lub czasem lub kolumny tabeli typu date lub timestamp:

Na przykład zdanie:

```
select nr_zawodnika
      , extract ( year from sysdate ) as "Bieżący rok"
      , extract ( year from data_urodzenia ) as "Rok urodzenia"
      , extract ( year from sysdate ) - extract ( year from data_urodzenia ) as "Wiek"
  from bd3_zawodnicy;
```

da wynik:

NR_ZAWODNIKA	Bieżący rok	Rok urodzenia	Wiek
263	2018	1975	43
264	2018	1977	41
265	2018	1966	52
266	2018	1970	48
.....			

Uwaga:

W środowisku Oracle nie jest możliwe użycie poniższej konstrukcji:

```
select nr_zawodnika
      , extract (year from sysdate) as "Bieżący rok"
      , extract (year from data_urodzenia) as "Rok urodzenia"
      , "Bieżący rok" - "Rok urodzenia" as "Wiek"
  from bd3_zawodnicy;
```

Nie można używać aliasów do obliczeń bezpośrednio w liście `select`, można ich używać tylko we frazie `order by`. Powyższy problem rozwiązuje się przy pomocy innej konstrukcji, która będzie omówiona w dalszej części materiału.

to_char:

Funkcja ta konwertuje wartości typu number, date lub timestamp do postaci znakowej. Najczęściej stosuje się ją do innej prezentacji danych z tabeli, na przykład, wymaganej na tworzonych raportach. Ogólna postać tej funkcji wygląda tak, jak poniżej:

```
to_char ( wartość_numeryczna_lub_data/czas , format )
```

gdzie:

`wartość_numeryczna_lub_data/czas` - kolumna tabeli, zmienna lub konkretna wartość typu number, date lub timestamp,

`format` - wzorzec docelowej postaci przekonwertowanych danych, do jego zdefiniowania w przypadku dat używa się takich symboli jak: YYYY lub YY, Q, MM, MONTH, MON, DD, DY, DAY.

Poniżej zostanie zaprezentowane użycie tej funkcji w stosunku do dat. Konwersja danych numerycznych nie będzie omawiana w tym materiale.

Przykładowo zdanie:

```
select nazwisko, data_urodzenia
      , to_char( data_urodzenia, 'DAY DD.MM.YYYY') as "Data urodzenia"
   from bd3_zawodnicy;
```

da wynik:

NAZWISKO	DATA_URODZENIA	Data urodzenia
Kuczkowski	75/05/26	PONIEDZIAŁEK 26.05.1975
Cieślak	77/07/22	PIĄTEK 22.07.1977
Strzałkowski	66/06/28	WTOREK 28.06.1966
Kowalski	70/09/17	CZWARTEK 17.09.1970
Grabowski	79/10/11	CZWARTEK 11.10.1979

.....

Funkcje konwersji, podobnie jak inne funkcje, mogą podlegać zagnieżdżaniu. Założymy, że chcemy datę w postaci napisu przekonwertować na inną postać napisu. Wykonanie zdania:

```
select to_char ( '24-02-1996', 'DD-MM-YYYY') as "Data urodzenia"
      from dual;
```

zakończy się niepowodzeniem, gdyż wprowadzana data nie jest w akceptowanym formacie:

```
select to_char ( '24-02-1996', 'DD-MM-YYYY') as "Data urodzenia"
      from dual
Error report -
ORA-01722: niepoprawna liczba
```

Należy najpierw z napisu zrobić datę, a następnie powtórnie dokonać konwersji według innego formatu, na przykład:

```
select to_char ( to_date ( '24-02-1996', 'DD-MM-YYYY'),
                  'fm DY DD MONTH YYYY') as "Data urodzenia"
      from dual;
```

Data urodzenia
SO 24 LUTY 1996

Prefix *fm* służy do wskazania, aby zbędne spacje w przekonwertowanej wartości zostały usunięte.

Uwaga:

Konwersja przy pomocy funkcji *to_char* w celu prezentacji w zakładanej formie danych numerycznych czy też dat straciła na znaczeniu w języku SQL z powodu powstania szeregu wyspecjalizowanych programów do tworzenia raportów. Mają one zaimplementowane, co najmniej, takie same funkcjonalności związane z konwersją. Przykładem może być JasperReports omawiany w dalszej części materiałów.

Funkcje ogólne

Istnieją funkcje działające na wszystkich typach danych i odnoszą się do używania wartości *NULL* w obliczeniach. Należą do nich funkcje:

- *nvl* (expr1, expr2) - konwertuje *NULL* w expr1 na określoną wartość (expr2),
- *nullif* (expr1, expr2) - porównuje oba wyrażenia i zwraca *NULL*, gdy są one równe, w przeciwnym przypadku zwraca pierwsze wyrażenie,
- *coalesce* (expr1, expr2,, exprn) - zwraca pierwsze wyrażenie z listy, które nie jest *NULL*.

Poniżej zostanie omówiona tylko pierwsza z tych funkcji, czyli *nvl*.

Jak już wcześniej zaznaczono - *NULL* ma tę właściwość, że działanie: wartość + *NULL* = *NULL*.

Wykonując zapytanie:

```
select nr_zawodnika, punkty_globalne, punkty_kategorii
from bd3_wyniki;
```

otrzymujemy zbiór, którego fragment jest przedstawiony poniżej:

NR_ZAWODNIKA	PUNKTY_GLOBALNE	PUNKTY_KATEGORII
849	2	35
850	(null)	(null)
851	(null)	(null)
853	(null)	18
855	(null)	10

Dla celów prezentacji działań na wartościach *NULL* rozszerzmy listę *select* o dodatkową kolumnę będącą sumą obu kolumn ze zdobytymi punktami⁴:

```
select nr_zawodnika, punkty_globalne, punkty_kategorii
      , punkty_globalne + punkty_kategorii as suma_pkt
from bd3_wyniki;
```

Otrzymamy:

NR_ZAWODNIKA	PUNKTY_GLOBALNE	PUNKTY_KATEGORII	SUMA_PKT
849	2	35	37
850	(null)	(null)	(null)
851	(null)	(null)	(null)
853	(null)	18	(null)
855	(null)	10	(null)

Wyniki są błędne dla dwóch wierszy. Zastosowanie funkcji *nvl* pozwala je poprawić.

Powyższe zdanie można zbudować tak:

```
select nr_zawodnika, punkty_globalne, punkty_kategorii
      , nvl( punkty_globalne, 0 ) + nvl( punkty_kategorii, 0 ) as suma_pkt
from bd3_wyniki;
```

, a wyniki ulegną zmianie:

⁴ W praktyce nie ma takiej klasyfikacji, w której należało dodawać te dwie wielkości. Należy traktować ten przykład jako demonstrację właściwości *NULL*.

NR_ZAWODNIKA	PUNKTY_GLOBALNE	PUNKTY_KATEGORII	SUMA_PKT
849	2	35	37
850	(null)	(null)	0
851	(null)	(null)	0
853	(null)	18	18
855	(null)	10	10

Argumentem funkcji *nvl* może być również kolumna tabeli typu date lub varchar2, na przykład:

.... *nvl* (data_zapisu, sysdate)

oznaczać będzie, że jeśli w kolumnie data_zapisu pojawi się *NULL* to będzie on konwertowany na bieżącą datę. Podobnie dla wartości typu varchar2:

....*nvl* (wartosc_towaru, 'Brak danych')

Uwaga:

Warto zastanawiać się nad koniecznością permanentnego stosowania funkcji *nvl* w przypadku, gdy kolumna numeryczna w tabeli podlegająca obliczeniom została zdefiniowana w zdaniu *create table...* jako mogąca przyjmować wartości *NULL*.

Funkcje grupujące w zdaniach SQL

Język SQL umożliwia dokonywanie obliczeń na zbiorze wierszy tabeli, wynikiem których może być jedna wartość (skalar) lub kolumna. Operacje te są możliwe dzięki funkcjom grupującym:

- *count* - zliczanie wierszy ,
- *sum* - sumowanie wartości w kolumnie,
- *avg* - obliczanie średniej z wartości w kolumnie,
- *min* - znajdowanie minimalnej wartości w kolumnie,
- *max* - znajdowanie maksymalnej wartości w kolumnie.

COUNT – funkcja ta różni się od pozostałych funkcji grupujących, ponieważ zamiast prowadzić obliczenia na wartościach kolumn, zwraca tylko liczbę wierszy w tabeli.

Poniższe zapytanie zwraca liczbę zawodników płci męskiej z tabeli BD3_ZAWODNICY:

```
select count ( * ) as "Mężczyźni"
from bd3_zawodnicy
where plec = 'M';      -- wynik: 641
```

Użycie znaku * jako argumentu funkcji spowoduje przeliczenie wszystkich wierszy w tabeli.

Jeśli zachodzi potrzeba wykluczenia z zapytania wierszy zawierających wartości *NULL* w jednej z kolumn, należy zamiast * wpisać nazwę tej wybranej kolumny:

```
select count ( nr_klubu ) as "Mężczyźni"
from bd3_zawodnicy
where plec = 'M';      -- wynik: 638
```

Co oznacza, że trzech mężczyzn nie należy do żadnego klubu.

Funkcji tej można także używać do określenia, ile różnych od siebie wartości pojawia się w interesującej nas kolumnie. Na przykład chcąc się dowiedzieć ile jest nienullowych wartości w kolumnie Pozycja w tabeli BD3_WYNIKI należy użyć zdania:

```
select count ( pozycja ) -- zliczane są tylko wystąpienia nienullowe
from bd3_wyniki
where nr_zawodow = 1;
```

natomiast zdanie:

```
select count ( distinct pozycja )
from bd3_wyniki
where nr_zawodow = 1;
```

zwróci liczbę różnych wartości występujących w kolumnie Pozycja.

Zdanie `select count (pozycja) as "Ile no_null",
count (distinct pozycja) as "Ile różnych no_null"`

```
from bd3_wyniki
where nr_zawodow = 1;
```

wykona poniższe obliczenia:

Ile no_null	Ile różnych no_null
274	2

Co oznacza, że w tabeli BD3_WYNIKI są 274 wyniki dla zawodów o numerze 1, ale tylko dwie różne wartości w kolumnie Pozycja.⁵

Przykłady użycia pozostałych funkcji:

```
select sum ( punkty_globalne ) as Suma_pkt_globalne
from bd3_wyniki w, bd3_zawodnicy z, bd3_kategorie k
where w.nr_zawodnika = z.nr_zawodnika
and z.nr_kategorii = k.nr_kategorii
and w.nr_zawodow=1
and k.nazwa_kategorii = 'K-III' ;
```

```
select round ( avg ( 2021 - extract ( year from data_urodzenia )), 1) as "Średni_wiek"
from bd3_zawodnicy
where nr_klubu = 2;
```

```
select max ( punkty_kategorii ) as Max_pkt_335
, min ( punkty_kategorii ) as Min_pkt_335
from bd3_wyniki
where nr_zawodnika = 335;
```

Należy zwrócić uwagę na działanie powyższych funkcji, w przypadku gdy w kolumnie będącej argumentem takiej funkcji występują wartości *NULL*.

Przy pomocy zapytania:

```
select nr_zawodnika, punkty_globalne
from bd3_wyniki
where nr_zawodnika = 934;
```

⁵ Kolumna *Pozycja* w tabeli BD3_WYNIKI symuluje działanie fotokomórki. Jeśli kilku zawodników osiągnęło na mecie ten sam czas to w kolumnie *Pozycja* określa się ich kolejność. Prezentowane wyniki każą wnioskować, że w tych zawodach zadziałała fotokomórka i w kolumnie *Pozycja* występują wartości: 1 i 2, a więc był co najmniej jeden przypadek, gdy dwóch zawodników przybiegły na metę w tym samym czasie, natomiast nie było przypadku, by trzech zawodników osiągnęło ten sam rezultat.

można się dowiedzieć jakie wyniki, w kontekście punktów globalnych, osiągnął konkretny zawodnik:

NR_ZAWODNIKA	PUNKTY_GLOBALNE
934	16
934	(null)
934	15

Zawodnik o numerze 934 startował trzy razy w sezonie, dwa razy zdobył punkty a raz nie.

Użycie dla powyższych danych funkcji agregujących przy pomocy zdania:

```
select max ( punkty_globalne ) as Max_pkt_934
      , min ( punkty_globalne ) as Min_pkt_934
      , sum ( punkty_globalne ) as Suma_pkt_934
      , avg ( punkty_globalne ) as Średnia_pkt_934
  from bd3_wyniki
 where nr_zawodnika = 934;
```

da poniższy zbiór wyników:

MAX_PKT_934	MIN_PKT_934	SUMA_PKT_934	ŚREDNIA_PKT_934
16	15	31	15,5

Uwagi:

1. Funkcje agregujące nie uwzględniają wartości *NULL* w realizacji obliczeń.
2. Interesujące są wyniki zwrócone przez funkcję *avg*. Zawodnik trzy razy brał udział w zawodach, ale średnia została policzona dla dwóch. Nie miejsce tutaj na roztrząsanie czy to jest prawidłowe czy nie. Trzeba postawić pytanie, czy można zastosować rozwiązanie uwzględniające ten trzeci bieg? Otóż można stosując funkcję *nvl*:

```
select avg ( punkty_globalne ) as Średnia_pkt_934_I
      , round ( avg ( nvl ( punkty_globalne , 0 ) ), 1) as Średnia_pkt_934_II
  from bd3_wyniki
 where nr_zawodnika = 934;
```

Otrzymamy inny wynik:

ŚREDNIA_PKT_934_I	ŚREDNIA_PKT_934_II
15,5	10,3

Który jest prawidłowy zależy od biznesowych założeń, ale widać, że język SQL jest przygotowany na obie ewentualności.

3. Można użyć do liczenia średniej konstrukcji *sum (kolumna) / count (kolumna)* zamiast funkcji *avg*. Należy tylko pamiętać o zasadach przedstawionych powyżej, szczególnie gdy kolumna będąca argumentem funkcji może zawierać wartości *NULL*.

Zapytania grupujące

Zapytania grupujące umożliwiają tworzenie macierzowych zbiorów wynikowych przy użyciu funkcji grupujących.

Aby poszgregować dane w grupy, należy dodać do zdania *select* frazę *group by* zawierającą nazwy kolumn, których wartości mają zostać użyte przy formowaniu grup.

Na przykład zdanie:

```
select nr_klubu as "Nr klubu", count( * ) as "Liczba zawodników"
from bd3_zawodnicy
group by nr_klubu
order by nr_klubu;
```

zwróci zestawienie obrazujące liczbę zawodników należących do poszczególnych klubów:

Nr klubu	Liczba zawodników
1	146
2	23
3	104
4	7
6	46
7	8
8	39
10	72
11	4

....

Należy bezwzględnie przestrzegać zasady, że w klauzuli *group by* muszą znaleźć się wszystkie kolumny poza funkcjami agregującymi, które występują na liście *select*.

W przypadku użycia zdania:

```
select nr_klubu as "Nr klubu", count(*) as "Liczba zawodników"
from bd3_zawodnicy
order by nr_klubu;
```

zostanie wygenerowany błąd:

```
Error report -
SQL Error: ORA-00937: to nie jest jednogrupowa funkcja grupowa
00937. 00000 -  "not a single-group group function"
*Cause:
*Action:
```

gdź brak jest frazy *group by* z kolumną *nr_klubu*.

Chcąc tworzyć bardziej pogłębione raporty analityczne można stosować zagnieżdzanie grup stosując konsekwentnie zasadę opisaną powyżej.

Poniższe zdanie obrazuje grupowanie dwupoziomowe zliczające zawodników w klubach w podziale na kategorie wiekowe:

```
select nazwa_kategorii, nazwa_klubu, count ( * ) as "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
join bd3_kategorie ka on z.nr_kategorii = ka.nr_kategorii
group by nazwa_klubu, nazwa_kategorii
order by nazwa_klubu, nazwa_kategorii;
```

Należy zwrócić uwagę na występowanie we frazie `select` obu kolumn, według których nastąpiło grupowanie.

NAZWA_KATEGORII	NAZWA_KLUBU	Liczba zawodników
VII	Bielanski KB Warszawa	1
II	Gwardia Warszawa	1
III	Gwardia Warszawa	5
K-III	Gwardia Warszawa	1
K-V	Gwardia Warszawa	2
V	Gwardia Warszawa	1
VI	Gwardia Warszawa	2
III	KB Amator Mińsk Mazowiecki	1
IV	KB Amator Mińsk Mazowiecki	1
V	KB Amator Mińsk Mazowiecki	1
VIII	KB Amator Mińsk Mazowiecki	1
III	KB Gymnasion Warszawa	20
IV	KB Gymnasion Warszawa	20

....

Omówione do tej pory zapytania grupujące wykorzystują do utworzenia grup wszystkie wiersze w tabeli bazowej. Istnieją sposoby zmniejszenia liczby uwzględnianych wierszy:

- Ograniczenie liczby wierszy przed sformowaniem grup – przez użycie frazy `where`:

```
select nazwa_klubu, count (*) as "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
where nazwa_klubu like '%Warszawa%'
group by nazwa_klubu
order by nazwa_klubu;
```

- Utworzenie grup, a następnie ograniczenie ich liczebności – przez użycie frazy `having`:

```
select nazwa_klubu, count ( * ) as "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
group by nazwa_klubu
having count ( * ) > 30
order by "Liczba zawodników" desc;
```

NAZWA_KLUBU	Liczba zawodników
Allianz Warszawa	146
KB Gymnasion Warszawa	104
KB Trucht Warszawa	100
KB Lechici Zielonka	73
AZS Uniwersytet Warszawski	72
KB Promyk Ciechanów	46
AZS SGGW Warszawa	39
KB Legionowo	32
Warszawianka	32

Do zbioru wynikowego zostaną wstawione tylko te nazwy klubów wraz z ich licznosciami, w których liczność zawodników przekracza 30.

Uwagi:

1. We frazie *group by* oraz *having* nie można używać aliasów, natomiast w *order by* - tak.
2. Kolejność występowania kolumn we frazie *group by* nie ma znaczenia, natomiast we frazie *order by* - tak.
3. W przypadku użycia konstrukcji grupujących nie jest możliwe sortowanie według kolumny, która nie występuje we frazie *group by* (nie ma jej na liście *select*).

Zadania do samodzielnego wykonania:

1. Opracować zestawienie zawodników zawierające imię i nazwisko jako jedną kolumnę o postaci A.ABACKI, wiek zawodnika i nazwę kategorii, dla zawodników, których numery zawarte są w przedziałach [30, 60] i [300,350]. Posortować zbiór malejąco według wieku zawodnika.
2. Obliczyć, ile kobiet należy do klubów o numerach z przedziału [10,20], które brały udział w zawodach nr 4.
3. Opracować zestawienie liczności uczestników zawodów nr 3 o postaci:

Nazwa kategorii, Liczba uczestników

Posortować malejąco zbiór według liczności uczestników.

4. Opracować zestawienie pokazujące sumaryczną liczbę zdobytych punktów w klasyfikacji generalnej (*punkty_globalne*) przez poszczególne kluby w zawodach nr 1 uwzględniając tylko mężczyzn. Raport zawierający numery klubów posortować malejąco według zdobytych punktów i numerów klubów rosnąco i nie pokazywać w nim klubów, które nie zdobyły punktów.
5. Opracować zestawienie pokazujące sumaryczną liczbę zdobytych punktów w klasyfikacji generalnej przez poszczególne kluby w zawodach nr 3 uwzględniając tylko kobiety. Raport ten zawierający nazwy klubów posortować malejąco według zdobytych punktów i nie pokazywać w nim klubów, które zdobyły mniej niż 50 punktów.
6. Opracować zestawienie obrazujące średni wiek kobiet i mężczyzn uczestniczących w poszczególnych zawodach o postaci:

Płeć, Data zawodów, Nazwa zawodów, Średni wiek, Liczba zawodników

Średni wiek przedstawić z dokładnością do jednego miejsca po przecinku (np. 42.4), a datę zawodów w formacie zawierającym pełną nazwę miesiąca. Zbiór wyników uporządkować według daty zawodów, płci oraz liczby zawodników malejąco.

7. Utworzyć tabelę mogącą przechowywać zagregowane dane dotyczące klubów o strukturze: nazwa klubu, liczności zarejestrowanych zawodników, średniej ich wieku. Wykorzystać konstrukcję *insert into table... select* .
8. Utworzyć tabelę mogącą przechowywać zagregowane dane dotyczące daty zawodów (o postaci YYYY-MM-DD) oraz liczby uczestniczących w nich zawodników. Wykorzystać konstrukcję *create table ...as select ...* .
9. Wykonać eksperyment polegający na uruchomieniu poniższych zdań zawierających funkcje agregujące:

```
select count ( * ) as "Liczba zawodników"  
from bd3_zawodnicy;
```

```
select nr_klubu as "Nr klubu", count ( * ) as "Liczba zawodników"  
from bd3_zawodnicy  
group by nr_klubu;
```

```
select count ( * ) as "Liczba zawodników"  
from bd3_zawodnicy  
group by nr_klubu;
```

```
select avg (count ( * )) as "Średnia liczba zawodników"  
from bd3_zawodnicy  
group by nr_klubu;
```

```
select max (count ( * )) as "Maks liczba zawodników"  
from bd3_zawodnicy  
group by nr_klubu;
```

Zastanowić się nad interpretacją wyników, szczególnie trzech ostatnich zdań. Jest to o tyle ważne, że takie konstrukcje występują w bardziej skomplikowanych zdaniach SQL, na przykład w podzapytaniach.

Bazy Danych laboratorium

**Laboratorium
BD5**

Na potrzeby zajęć zostanie wykorzystany model bazy danych opisany i zaimplementowany w ramach Laboratorium BD3.

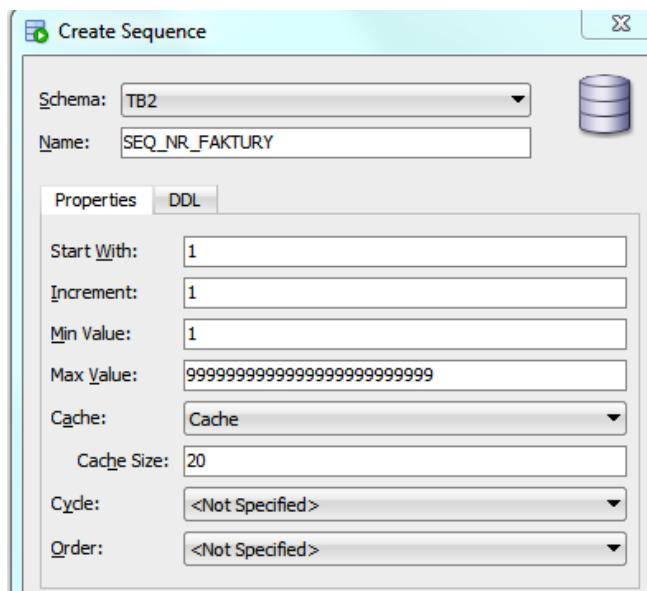
Poza podstawowymi obiektami bazodanowymi jakimi są tabele ze swoimi strukturami i relacjami, można tworzyć i wykorzystywać w projektowaniu baz danych inne obiekty. Poniżej zostaną zaprezentowane trzy rodzaje takich obiektów: sekwencje, indeksy i perspektywy oraz dodatkowo zmienne wiążania.

Tworzenie i zastosowanie sekwencji

Dużym problemem przy projektowaniu aplikacji bazodanowych jest zapewnienie niepowtarzalności wartości klucza głównego w tabeli. Typową sytuacją jest tabela zawierająca pewną liczbę wierszy, w której klucz główny przyjmuje kolejne wartości liczbowe (na przykład tabela faktur). Chcemy wprowadzić zdaniem *insert* wiersz zawierający na pozycji klucza głównego kolejną numeryczną wartość lub też zrealizować to poprzez formularz w aplikacji. Teoretycznie, najpierw trzeba by było odpytać tabelę o maksymalną wartość klucza głównego i na tej podstawie określić wartość bieżącą. Przy pomocy sekwencji ten proces można uprościć.

Sekwencja jest trwałym obiektem bazodanowym tworzonym bądź to poprzez kreator w Oracle SQL Developer bądź zdaniem *create sequence*.

Typowymi wartościami parametrów sekwencji są:



W wyniku takiego ustawienia parametrów można wygenerować sekwencję równoważną zdaniu SQL:

```
create sequence seq_nr_faktury
increment by 1
start with 1
maxvalue 999999999999999999999999999999
minvalue 1
cache 20;
```

Najprostszym zdaniem tworzącym sekwencję jest:

```
create sequence seq_nr_faktury;
```

Tak zdefiniowana sekwencja przyjmuje standardowe wartości, które zostały przedstawione na powyższym rysunku lub:

```
create sequence seq_nr_faktury nocache;
```

, co oznacza brak cachowania kolejnych generowanych wartości klucza głównego, co może spowalniać proces ładowania danych do hurtowni danych.

W celu skasowania sekwencji należy użyć zdania *drop*, np.:

```
drop sequence seq_nr_faktury;
```

Definicję sekwencji można modyfikować zdaniem *alter sequence.....*, na przykład:

```
alter sequence seq_nr_faktury increment by 2;
```

, ale w ten sam sposób nie można zmienić frazy *start with*:¹

```
alter sequence seq_nr_faktury start with 3000;
```

Wykonanie tego zdania spowoduje błąd:

```
Error report -
ORA-02283: nie można zmienić początkowego numeru sekwencji
02283. 00000 - "cannot alter starting sequence number"
```

Z każdą sekwencją związane są dwie wartości (pseudokolumny): *current value* (*currval*) i *next value* (*nextval*). Pierwsza z nich wskazuje ostatnią wygenerowaną wartość sekwencji (w danej sesji), a druga - wartość, która zostanie wygenerowana przy najbliższym uruchomieniu sekwencji bez względu na sesję. Wartości te można odczytać poprzez użycie zdania *select*:

```
select seq_nr_faktury.nextval from dual; -- wygenerowana jest nowa wartość
select seq_nr_faktury.currval from dual;2 -- odczytana jest bieżąca wartość
```

Przykłady zastosowania sekwencji:

```
create sequence seq_nr_wpisu; -- utworzenie sekwencji
```

```
create table testowa    -- utworzenie tabeli testowej
(
  nr_wpisu number (5) primary key,
  tresc varchar2 (120)
);
insert into testowa (nr_wpisu, tresc) -- wprowadzenie pierwszego wiersza do tabeli
values
(
  seq_nr_wpisu.nextval,
  'Pierwszy wpis testowy'
);
```

¹ Można tej modyfikacji dokonać wykorzystując możliwość zmiany inkrementacji w sekwencji.

² Wartość *current value* jest określona dopiero po pierwszym odwołaniu się do sekwencji czyli wygenerowaniu wartości *next value*.

```

insert into testowa (nr_wpisu, trecs) -- wprowadzenie drugiego wiersza do tabeli
values
(
    seq_nr_wpisu.nextval,
    'Drugi wpis testowy'
);

commit;          -- zatwierdzenie transakcji

```

Zdania `select` zwrócią poniższe wartości:

```
select * from testowa;
```

NR_WPISU	TRESC
1	Pierwszy wpis testowy
2	Drugi wpis testowy

```
select 'current value' as " ", seq_nr_wpisu.currval as "value"
from dual;
```

	value
current value	2

```
select 'next value' as " ", seq_nr_wpisu.nextval as "value"
from dual;
```

	value
next value	3

Powyższy przykład można zmodyfikować zmieniając definicję klucza głównego w tabeli:

```
alter table testowa
modify nr_wpisu default seq_nr_wpisu.nextval;
```

oraz zdanie `insert`:

```

insert into testowa (trecs) -- wprowadzenie kolejnego wiersza do tabeli
values
(
    'Trzeci wpis testowy'
);

```

Odpytanie tabeli da wynik:

NR_WPISU	TRESC
1	Pierwszy wpis testowy
2	Drugi wpis testowy
3	Trzeci wpis testowy

Numer 3 został "zużyty" w zdaniu `select` zwracającym następną wartość sekwencji w jednym z powyższych zdań.

Tworzenie i zastosowanie indeksów

Indeks jest strukturą danych umożliwiającą szybki dostęp do wybranych wierszy tabeli bazowej w oparciu o wartości jednej lub większej liczby kolumn (klucz indeksu). Tabela bazowa nie musi być uporządkowana, to indeks powoduje, że w zbiorze wyników wiersze pojawiają się według pewnego porządku.

Indeks umożliwia przeglądanie zbioru bazowego szybciej. Optymalizator zapytań każdorazowo decyduje czy wykorzystanie gotowego indeksu przyspieszy aktualną operację i w razie potrzeby odwoła się do jego zawartości.

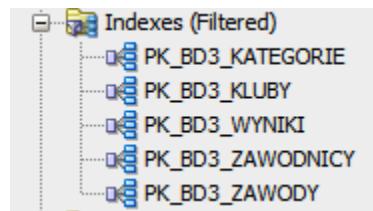
Jeżeli tabela bazowa ma przypisany klucz główny (Primary Key) system zarządzania bazą danych (SZBD) automatycznie tworzy indeks tabeli w oparciu o ten klucz. W momencie wprowadzania nowego wiersza, pierwszą czynnością wykonywaną przez SZBD jest sprawdzenie, czy klucz główny tego wiersza występuje już w indeksie danej tabeli – tak więc unikatowość wierszy w rzeczywistości jest kontrolowana na poziomie indeksu, a nie na poziomie tabeli bazowej.

Nad tworzeniem indeksów opartych o klucze główne nie mamy kontroli – SZBD utworzy je automatycznie.³ Możemy jednak dodatkowo tworzyć inne indeksy pamiętając, że każdy z nich to nowa struktura.

Indeksy powodują:

- Zwiększenie bazy danych,
- Spowolnienie operacji wprowadzania, modyfikacji i usuwania wierszy z tabeli, gdyż SZBD musi zaktualizować wszystkie struktury indeksów związanych z tabelą bazową,
- Przyspieszenie pozyskiwania danych z tabel.

Strukturę indeksów można oglądać z poziomu Oracle SQL Developer rozwijając i ewentualnie filtrując folder *Indexes*:

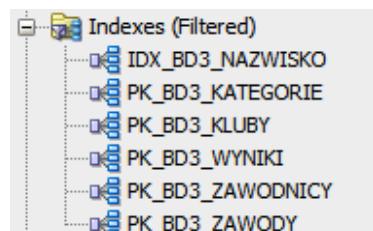


Powyższe indeksy powstały automatycznie w momencie tworzenia tabel ze zdefiniowanymi kluczami głównymi.

Istnieje możliwość zapoznania się ze szczegółowymi właściwościami danego indeksu poprzez wskazanie go i wybieranie poszczególnych zakładek (głównie Columns i SQL). Warto porównać budowę indeksu PK_BD3_ZAWODNICY (oparty o jedną kolumnę) oraz PK_BD3_WYNIKI (oparty o dwie kolumny).

Indeksy tworzy się zdaniem *create index*, np.:

```
create index idx_bd3_nazwisko on bd3_zawodnicy (nazwisko);
```



³ Istnieją SZBD, na przykład Sybase, które automatycznie tworzą indeksy związane z kluczami obcymi oprócz indeksów opartych na kluczu głównym. SZBD Oracle nie posiada takiej właściwości.

Należy porównać pozycję UNIQUENESS znajdującą się w zakładce Details dla nowoutworzonego indeksu oraz dla indeksów opartych o klucz główny. Indeks oparty o nazwisko nie jest unikalny, bo nie jest kluczem głównym - nic nie stoi na przeszkodzie, aby kilku zawodników miało to samo nazwisko.

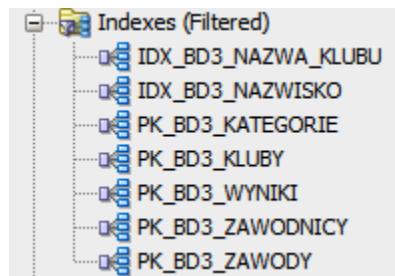
W przypadku użycia zdania:

```
select nazwisko, imie
from bd3_zawodnicy
where nazwisko like 'Kowal%'
order by nazwisko;
```

można spodziewać się, że optymalizator zapytań SQL automatycznie dobierze indeks *idx_bd3_nazwisko* w celu utworzenia zbioru wyników, co przyspiesza przetwarzanie.

Indeksy unikalne tworzone są zdaniem:

```
create unique index idx_bd3_nazwa_klubu on bd3_kluby (nazwa_klubu);
```



Poprzez indeks unikalny można kontrolować, czy nazwa klubu podana przy nowowprowadzanym klubie do ewidencji już istnieje. Można wykonać test polegający na próbie wprowadzenia jako nowego, klubu już zarejestrowanego, na przykład:

```
insert into bd3_kluby values (50, 'KU AZS WAT Warszawa');
```

Próba zakończy się niepowodzeniem, gdyż klub o tej nazwie już jest zarejestrowany.

Indeksy unikalne tworzone są automatycznie w przypadku definiowania kolumny tabeli jako *unique*:

```
alter table bd3_zawody
    modify nazwa_zawodow constraint idx_bd3_nazwa_zawodow unique;
```

Usuwanie indeksów wykonuje się zdaniem *drop*:

```
drop index idx_bd3_data_urodzenia;
```

A w przypadku, gdy powstał poprzez definicję constraint (na przykład *idx_bd3_nazwa_zawodow*) należy usunąć ten constraint:

```
alter table bd3_zawody
    drop constraint idx_bd3_nazwa_zawodow;
```

Pozostawia się do samodzielnej obserwacji istnienie indeksów w przypadku kasowania tabeli bazowej.

Uwagi:

1. Indeksy pełnią bardzo ważną rolę w zwiększeniu wydajności serwerów bazodanowych, dlatego są niezbędne w zarządzaniu i eksploatacji hurtowni danych,

2. Badanie wydajności baz danych nie jest oparte o ścisłe prawa czy zależności. Po części jest to proces eksperymentalny i jego efekty zależą od konfiguracji konkretnej bazy, wiedzy i doświadczenia analityka czy administratora bazy,
3. W kontekście stosowania indeksów istnieje kilka reguł, które są skuteczne w większości przypadków ich stosowania. Tworzyć należy indeksy oparte o:
 - klucze główne (tworzone automatycznie),
 - klucze obce (istotne przy łączeniu tabel),
 - kolumny występujące we frazie *where* (filtrowanie wierszy).
4. Przy badaniu sprawności wykonywania się poszczególnych zdań *select* korzysta się z planów wykonania (*executon plan*, *explain plan*).

Poniżej przedstawiony zostanie poglądowy scenariusz takiego działania:

- skasować indeks *idx_bd3_nazwisko* (jeśli istnieje),
- wykonać zdanie *select* (w celu sprawdzenia jego poprawności):

```
select nazwisko
from bd3_zawodnicy
where nazwisko = 'Kowalski';
```

- ustawić kurSOR myszy na terenie tego zdania i wcisnąć przycisk F10:

OBJECT_NAME	OPTIONS	CARDINALITY	COST
BD3_ZAWODNICY	FULL	5	3
		5	3

optymalizator ustalił, że to zdanie będzie się wykonywało poprzez pełen przegląd tabeli (*full*) i koszt wykonania wynosi 3,

- założyć indeks w oparciu o kolumnę nazwisko,

```
create index idx_bd3_nazwisko on bd3_zawodnicy (nazwisko);
```

- wygenerować plan wykonania dla zdania:

```
select nazwisko
from bd3_zawodnicy
where nazwisko = 'Kowalski';
```

OBJECT_NAME	OPTIONS	CARDINALITY	COST
IDX_BD3_NAZWISKO	RANGE SCAN	5	2
		5	2

ten *select* wcale nie będzie odczytywał danych z tabeli, gdyż nazwiska są zawarte w indeksie. Koszt wykonania jest niższy.

- skasować utworzony indeks:

```
drop index idx_bd3_nazwisko;
```

Tworzenie i zastosowanie perspektyw

Perspektywa (*view*) jest tabelą wirtualną (logiczną) tworzoną zdaniem *create view...*. Ma ona własną nazwę i jest przechowywana w bazie danych z tą różnicą w stosunku do tabeli fizycznej, że brak w niej danych.

Perspektywy używa się z kilku powodów:

- Umożliwiają zapisanie często wykonywanych złożonych zapytań w strukturze bazy. Można wówczas użyć wyrażenia:

```
select * from nazwa_perspektywy;
```

To zwalnia nas z konieczności każdorazowego wpisywania całej formuły zapytania.

- Pomagają w dostosowaniu środowiska bazodanowego do indywidualnych potrzeb użytkowników lub ich grup tworząc dedykowane zapytania,
- Umożliwiają zapewnienie bezpieczeństwa danych. Zamiast nadawać użytkownikom prawa wglądu w całą strukturę tabeli, tworzona jest perspektywa zawierającą tylko te dane, które rzeczywiście są potrzebne, po czym przyznawane są prawa dostępu do wybranej perspektywy.

Perspektywy zawierają zawsze aktualne dane, gdyż ich zawartość jest tworzona każdorazowo przy wywołaniu na podstawie danych z tabel bazowych.

Perspektywa tworzona jest zdaniem *create view* według schematu:

```
create or replace view nazwa_perspektywy as
select .....
```

Przykładowo, aby utworzyć perspektywę zawierającą wybrane dane z ewidencji zawodników, można napisać poniższe zdanie:

```
create or replace view bd3_ewidencja as
select nazwisko || ' ' || imie as "Zawodnik", nazwa_klubu as "Klub"
from bd3_zawodnicy z join bd3_kluby k
on z.nr_klubu = k.nr_klubu;
```

a następnie użyć jej w prostym zapytaniu:

```
select * from bd3_ewidencja
order by "Zawodnik";
```

które da taki wynik:

Zawodnik	Klub
Adach Krzysztof	Allianz Warszawa
Adamczyk Marcin	KB Lechici Zielonka
Adamski Wojciech	KB Promyk Ciechanów
Alabrudziński Jerzy	Allianz Warszawa
Anc Michał	KB Lechici Zielonka
Anczewski Leszek	KB Trucht Warszawa
Anders Andrzej	KB Orientuz Warszawa
.....	

Można w definicji perspektywy nadawać kolumnom własne nazwy, np.:

```
create or replace view bd3_ewidencja ( "Zawodnik" , "Klub" ) as
    select nazwisko || ' ' || imie, nazwa_klubu
        from bd3_zawodnicy z join bd3_kluby k
        on z.nr_klubu = k.nr_klubu;
```

i wtedy zdanie

```
select * from bd3_ewidencja
order by "Zawodnik";
```

da taki wynik:

Zawodnik	Klub
Adach Krzysztof	Allianz Warszawa
Adamczyk Marcin	KB Lechici Zielonka
Adamski Wojciech	KB Promyk Ciechanów
Alabrudziński Jerzy	Allianz Warszawa
Anc Michał	KB Lechici Zielonka
Anczewski Leszek	KB Trucht Warszawa
Anders Andrzej	KB Orientuz Warszawa

.....

Przy tworzeniu perspektyw można korzystać z dowolnych zdań `select` – również takich, które zawierająłączenia (powyższy przykład), unie i funkcje agregujące, jak również można zagnieżdżać perspektywy.

Przykładowo:

1. `create or replace view bd3_liczba_zawodnikow as
 select count(*) as "Liczba"
 from bd3_zawodnicy;`

Wydanie polecenia:

```
select * from bd3_liczba_zawodnikow;
```

da wynik:

Liczba
771

2. `create or replace view bd3_liczenie_punktow as
 select nr_zawodow, nr_zawodnika, rezultat_min, rezultat_sek,
 punkty_globalne as PktG,
 punkty_kategorii as PktK
 from bd3_wyniki;`

```
create or replace view bd3_sumy_pkt as
select nr_zawodnika, sum(PktG) as PktGlob, sum(PktK) as PktKat
from bd3_liczenie_punktow
group by nr_zawodnika;
```

Wydanie polecenia:

```
select *
  from bd3_sumy_pkt
 where PktGlob > 180
 order by PktGlob;
```

da wynik:

NR_ZAWODNIKA	PKTGLOB	PKTKAT
616	185	200
333	187	200
840	189	192
173	192	200
224	194	196
60	198	200

W celu usunięcia perspektywy z bazy należy użyć zdania *drop view nazwa_perspektywy*:

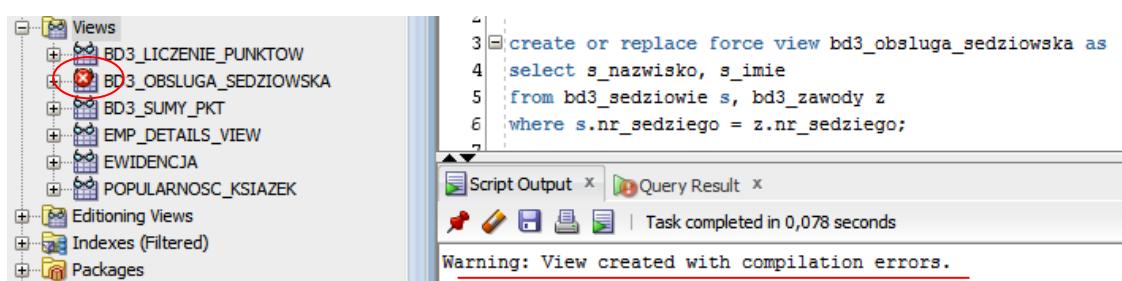
Np.: *drop view bd3_ewidencja;*

Uwagi:

1. W Oracle SQL Developer można obserwować tworzenie i usuwanie perspektyw wybierając folder Views,
2. Należy zauważać, że definicje perspektyw są zapamiętane w bazie po zamknięciu sesji (na stałe).
3. Tworzona perspektywa może mieć nadaną właściwość *force*:

```
create or replace force view bd3_obsługa_sedziowska as
select s_nazwisko, s_imie
  from bd3_sedziowie s, bd3_zawody z
 where s.nr_sedziego = z.nr_sedziego;
```

W schemacie brak jest tabeli BD3_SEDZIOWIE oraz brak jest kolumny nr_sedziego w tabeli BD3_ZAWODY. Klauzula *force* spowoduje, że perspektywa zostanie założona, chociaż będzie wykazany błąd.



Właściwość ta może być wykorzystana w pracach projektowych (na przykład zespołowych), w trakcie których można projektować fragmenty systemu w oparciu o istniejące nazwy perspektyw, chociaż brak jest implementacji ostatecznych definicji tabel.

4. W przypadku tworzenia perspektywy na podstawie zdania `select` zawierającego funkcję agregującą warto zwrócić uwagę na zachowanie się agregowanej wartości. Założmy, że opracowane jest zdanie `select` z funkcją agregującą, na przykład:

```
select nazwa_klubu "Klub", sum ( punkty_globalne ) "Punktacja"
from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
group by nazwa_klubu;
```

Chcąc, otrzymany na podstawie tego zdania, zbiór wynikowy ograniczyć należy zastosować frazę `having`:

```
.....
having sum ( punkty_globalne ) > 1000
```

Próba użycia do tego filtrowania frazy `where` zakończy się niepowodzeniem:

```
SQL Error: ORA-00934: funkcja grupowa nie jest tutaj dozwolona
00934. 00000 - "group function is not allowed here"
```

Po zbudowaniu na podstawie tego zdania perspektywy:

```
create or replace view bd3_punktacja_klubowa ( klub, punktacja ) as
select nazwa_klubu, sum ( punkty_globalne )
from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
group by nazwa_klubu;
```

i próbie użycia frazy `having` przy jej uruchomieniu:

```
select * from bd3_punktacja_klubowa
having punktacja > 1000
order by punktacja desc;
```

wykazany zostanie błąd:

```
SQL Error: ORA-00979: to nie jest wyrażenie GROUP BY
00979. 00000 - "not a GROUP BY expression"
```

, natomiast użycie frazy `where` zamiast `having` zakończy się powodzeniem:

KLUB	PUNKTACJA
KB Trucht Warszawa	2607
KB Gymnasion Warszawa	1763
Allianz Warszawa	1425

Kolumna będąca zbiorem agregatów w zdaniu `select` została niejawnie przekształcona w kolumnę będącą zbiorem wielkości skalarnych.

5. Perspektyw można używać w zdaniach DML (`insert`, `update`, `delete`), ale w ograniczonym zakresie. Do najważniejszych ograniczeń należą:

- nie może być oparta na wielu tabelach,
- nie może zawierać klauzuli `distinct`,
- nie może zawierać funkcji grupujących ani klauzuli `group by`,
- nie może zawierać wyrażeń w kolumnie.

Zastosowanie zmiennych wiązania w języku SQL

Zmienne wiązania (*bind variable*) mają, między innymi, zastosowanie w przekazywaniu danych między środowiskami programistycznymi, na przykład między językiem SQL i java, php, C itp. Innym zastosowaniem jest wprowadzanie danych do zdań SQL w pracy interaktywnej w Oracle SQL Developer lub SQL Plus.

Załóżmy, że chcemy testować zdanie *select* postaci:

```
select nazwisko || ' ' || imie "Zawodnik", nazwa_klubu "Klub"
from bd3_zawodnicy z join bd3_kluby k
on z.nr_klubu = k.nr_klubu
where k.nr_klubu = xxx ;           -- xxx - numer klubu wprowadzany
                                    -- interaktywnie
```

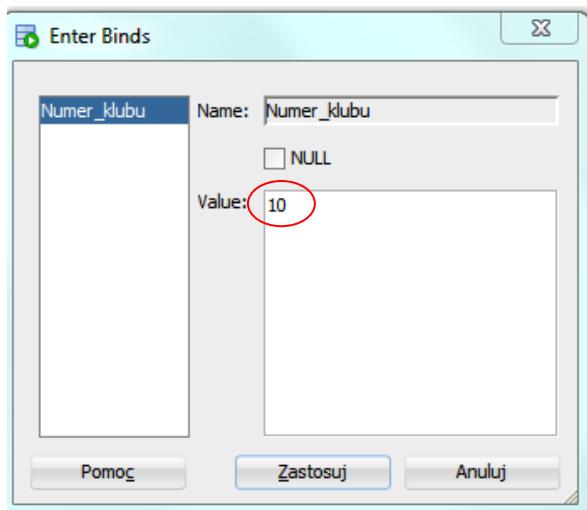
Oczywiście, że nic nie stoi na przeszkodzie, aby zmieniać interaktywnie wartość xxx przez nadpisanie. Ale można również to zdanie sparametryzować, na przykład:

```
select nazwisko || ' ' || imie "Zawodnik", nazwa_klubu "Klub"
from bd3_zawodnicy z join bd3_kluby k
on z.nr_klubu = k.nr_klubu
where k.nr_klubu = :Numer_klubu;
```

gdzie zmienna Numer_klubu (poprzedzona znakiem :) jest zmienną wiązania.

W tym przypadku zmienna ta nie musi być deklarowana oraz może mieć dowolną nazwę, na przykład :NR lub :Podaj_Numer_Klubu.

Po uruchomieniu tak skonstruowanego zdania w Oracle SQL Developer (F5) wyświetlony zostanie formularz:



, na którym należy w polu Value wprowadzić konkretną wartość, w tym przypadku numer klubu, zawodników którego ma dotyczyć tworzony zbiór wynikowy.

Jednocześnie można definiować wiele zmiennych wiązania, na przykład uruchomienie zdania:

```
select nazwisko || ' ' || imie "Zawodnik", nazwa_klubu "Klub"
from bd3_zawodnicy z join bd3_kluby k
on z.nr_klubu = k.nr_klubu
where z.nr_klubu = :Numer_klubu
      and data_urodzenia between :min_Data_urodzenia and :max_Data_urodzenia;
```

spowoduje wyświetlenie formularza, na którym należy wprowadzić wartości dla wszystkich zmiennych wiążania. Przykładowo powyższe zdanie dla:

```
min_Data_urodzenia = 1973/01/01          -- bez apostrofów
max_Data_urodzenia = 1973/01/31
Numer_klubu = 10
```

utworzy zbiór wynikowy:

Zawodnik	Klub
Mioduszewski Mateusz	AZS Uniwersytet Warszawski

, a zdanie:

```
select nazwisko || ' ' || imie "Zawodnik", nazwa_klubu "Klub"
from bd3_zawodnicy z join bd3_kluby k
on z.nr_klubu = k.nr_klubu
where z.nr_klubu = :Numer_klubu
and nazwisko like :bind_nazwisko;
```

dla danych:

```
Numer_klubu = 10
bind_nazwisko = %Ma%          -- bez apostrofów
```

utworzy zbiór:

Zawodnik	Klub
Mazurczyk Rafał	AZS Uniwersytet Warszawski
Marciniak Andrzej	AZS Uniwersytet Warszawski
Marcinkowski Tomasz	AZS Uniwersytet Warszawski

Uwaga:

Według tej samej zasady odbywa się przekazywanie danych w aplikacjach bazodanowych. Pola na formularzu aplikacji są zmiennymi wiązania i po jego wypełnieniu przenoszone są do zdania SQL, na przykład:

```
insert into ewidencja (nazwisko, imie, pesel, adres_zamieszkania)
values (:P10_NAZWISKO, :P10_IMIE, :P10_PESEL, :P10_ADRES);
```

gdzie zmienne z prefiksem P10 są nazwami pól na formularzu aplikacji.

Zadania do samodzielnego wykonania:

- Opracować perspektywę *bd3_liczba_zawodnikow* zliczającą zarejestrowanych zawodników w rozbiciu na płeć, a następnie na jej podstawie obliczyć liczbę zaewidencjonowanych zawodniczek.
- Opracować perspektywę *bd3_ewidencja* zawierającą kolumny:

Nazwisko, Klub, Kategoria, Wynik, Zawody, Data zawodów

przy czym:

Nazwisko jest złożeniem kolumn *nazwisko* i *imie*,
Klub to *nazwa_klubu*,
Kategoria to *nazwa_kategorii*,
Wynik to złożenie kolumn *rezultat_min* i *rezultat_sek* i przedstawienie jej w formacie
mm:ss, (wykorzystać funkcję *to_char* lub *cast*),
Zawody to kolumna *nazwa_zawodow*,
Data Zawodów to kolumna *data_zawodow*.

Opracować zdanie SQL, które na podstawie tej perspektywy będzie generować listę osiągnięć podanego tylko z nazwiska zawodnika porządkując zbiór malejąco według daty zawodów.

3. Założyć tabelę BD3_KLASYFIKACJA_GEN przy pomocy konstrukcji *create table as select...* przechowującą nr klubu, sumę zdobytych punktów w klasyfikacji generalnej, liczbę zawodników każdego klubu startujących w zawodach oraz liczbę zawodników zaewidencjonowanych w każdym klubie. Na jej podstawie opracować perspektywę generującą raport pokazujący klasyfikację klubów, w którym są uwzględnione tylko te kluby, które zdobyły punkty w tej klasyfikacji zawierający nazwę klubu, sumę zdobytych punktów w klasyfikacji globalnej oraz liczbę startujących zawodników.
Należy wykorzystać technikę złączeń zewnętrznych oraz odpowiednio zastosować argument funkcji *count*.
4. Założyć sekwencję umożliwiającą wprowadzanie nowych wierszy do tabeli BD3_ZAWODY przy uwzględnieniu już istniejących.
5. Założyć indeks zwiększający wydajność serwera dla wybranych kluczy obcych oraz dla wybranych kolumn często stosowanych przy filtrowaniu bazy (na przykład nazwisko i imię).

	<i>Bazy Danych laboratorium</i>	Laboratorium BD6
--	-------------------------------------	---------------------

Na potrzeby zajęć zostanie wykorzystany model bazy danych opisany i zaimplementowany w ramach Laboratorium BD3.

Poniżej omówione zostaną dwie zaawansowane techniki tworzenia raportów przy pomocy zdania `select` - zapytania krzyżowe (*pivot query*) mające zastosowanie w analityce biznesowej oraz podzapytania (*subquery*) zwiększące dynamikę opracowywanych zapytań.

Tworzenie i zastosowanie wyrażenia CASE

Wyrażenie CASE w języku SQL zastępuje instrukcję warunkową `if...then...else...endif`, i może mieć jedną z dwóch postaci:

Pierwsza postać:

```
case kolumna_w_tabeli
    when wartość_1 then wynik_1
    when wartość_2 then wynik_2
    .....
    when wartość_n then wynik_n
else wynik_inny
end;
```

np. zdanie:

```
select nr_zawodnika as "Zawodnik",
       ( case plec
           when 'K' then 'Kobieta'
           when 'M' then 'Mężczyzna'
           else 'Brak danych'
           end ) as "Płeć"
  from bd3_zawodnicy
 where nr_zawodnika in ( 34, 35, 500, 501 );
```

tworzy zbiór wynikowy:

Zawodnik	Płeć
34	Kobieta
35	Kobieta
500	Mężczyzna
501	Mężczyzna

Druga postać:

```
case
    when zdanie_logiczne_1 then wynik_1
    when zdanie_logiczne_2 then wynik_2
    .....
    when zdanie_logiczne_n then wynik_n
else wynik_inny
end;
```

np. zdanie:

```
select nr_zawodnika as zawodnik, rezultat_min, rezultat_sek,
       ( case
           when rezultat_min < 35 then 'Mistrzowska'
           when rezultat_min < 38 then 'I klasa'
           when rezultat_min < 40 then 'II klasa'
           else Null
           end ) as klasa_sportowa
  from bd3_wyniki
 where nr_zawodow = 1 and nr_zawodnika in ( 616, 636, 577, 34 )
 order by rezultat_min, rezultat_sek;
```

daje poniższe wyniki:

ZAWODNIK	REZULTAT_MIN	REZULTAT_SEK	KLASA_SPORTOWA
616	33	38	Mistrzowska
577	36	0	I klasa
636	39	11	II klasa
34	51	52	(null)

W języku SQL w wydaniu Oracle pierwsza postać wyrażenia case może być zastąpiona wyrażeniem decode, którego postać jest następująca:

```
... , decode ( kolumna,
               wartość_1, wynik_1,
               wartość_2, wynik_2,
               .....
               wartość_n wynik_n,
               wynik_inny ), ...
```

Porównywane są wartości pola w kolumnie z kolejnymi wartościami *wartość_...* i w przypadku równości zwracany jest odpowiedni *wynik_...*. Jeśli *wynik_inny* jest pominięty to zwracany jest *NULL*.

Powyższy przykład z pierwszą postacią case może być przedstawiony tak:

```
select nr_zawodnika as "Zawodnik",
       decode( plec ,
               'K' , 'Kobieta',
               'M' , 'Mężczyzna',
               'Brak danych') as "Płeć"
  from bd3_zawodnicy
 where nr_zawodnika in ( 34, 35, 500, 501 );
```

i daje ten sam wynik:

Zawodnik	Płeć
34	Kobieta
35	Kobieta
500	Mężczyzna
501	Mężczyzna

Wyrażenia case lub decode można użyć do budowy zdań SQL umożliwiających otrzymywanie raportów krzyżowych. Przykładem niech będzie poniższy raport pokazujący liczbę zawodników startujących w poszczególnych zawodach według kategorii wiekowych:

```

select z.nr_kategorii as "Nr kategorii", nazwa_kategorii as "Kategoria",
       sum( decode( nr_zawodow , 1 , 1 , 0 )) as "Zawody I",
       sum( decode( nr_zawodow , 2 , 1 , 0 )) as "Zawody II",
       sum( decode( nr_zawodow , 3 , 1 , 0 )) as "Zawody III",
       sum( decode( nr_zawodow , 4 , 1 , 0 )) as "Zawody IV",
       count(*) as "Łącznie"
from bd3_zawodnicy z join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
join bd3_kategorie k on k.nr_kategorii = z.nr_kategorii
group by z.nr_kategorii, nazwa_kategorii

union

select null, 'Razem',
       sum( decode( nr_zawodow , 1 , 1 , 0 )),
       sum( decode( nr_zawodow , 2 , 1 , 0 )),
       sum( decode( nr_zawodow , 3 , 1 , 0 )),
       sum( decode( nr_zawodow , 4 , 1 , 0 )),
       count(*)
from bd3_zawodnicy z join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
join bd3_kategorie k on k.nr_kategorii = z.nr_kategorii

order by 1;

```

Nr kategorii	Kategoria	Zawody I	Zawody II	Zawody III	Zawody IV	Łącznie
2 II		8	3	3	5	19
3 III		54	28	29	31	142
4 IV		56	45	38	61	200
5 V		40	31	27	29	127
6 VI		45	36	36	30	147
7 VII		10	10	5	3	28
8 VIII		6	5	4	2	17
9 IX		3	1	0	1	5
10 X		0	1	0	0	1
20 K-I		0	0	1	0	1
21 K-II		7	5	3	3	18
22 K-III		17	5	7	6	35
23 K-IV		12	7	9	14	42
24 K-V		7	6	10	5	28
25 K-VI		8	8	4	6	26
26 K-VII		1	0	1	0	2
(null) Razem		274	191	177	196	838

Należy zwrócić uwagę, że raport posiada dwa podsumowania: według zawodów i według kategorii wiekowych.

Uwagi:

1. Przy budowie raportów krzyżowych należy pamiętać, że każda nowa kolumna powstaje poprzez użycie niezależnego wyrażenia *decode* oraz funkcji agregującej,
2. Podsumowanie kolumn (stopka raportu) powstaje poprzez połączenie *union*, w którym liczba kolumn musi być taka sama jak w pierwszym zdaniu *select*,
3. Klucze sortowania odnoszą się do pierwszego zdania *select*.
4. W drugim zdaniu *select* na liście brak jest kolumn z tabeli (znajduje się tam tylko *Null* i napis 'Razem') i dlatego mimo zastosowania funkcji agregującej nie ma potrzeby użycia klauzuli *group by*.
5. Ta konstrukcja ma wadę polegającą na statycznym układzie kolumn. Przedstawione powyżej przykłady zapytania krzyżowego realizującego raport liczby startujących zawodników w podziale na kategorie wiekowe i numery zawodów zawierają stałą liczbę kolumn obrazującą analizę dotyczącą czterech zawodów. Zapytanie to nie będzie uwzględniało liczności zawodników w przypadku zarejestrowania wyników kolejnego piątego i dalszych zawodów.
6. Język SQL został poszerzony o frazę *pivot*, przy pomocy której możliwe jest wygenerowanie raportu krzyżowego. Poniżej przedstawiony został przykład użycia tej frazy przy tworzeniu raportu obrazującego sumaryczne zdobyczce punktowe klubów w klasyfikacji generalnej:

```

select * from
(
    select k.nr_klubu "Nr klubu",
           k.nazwa_klubu "Klub",
           w.nr_zawodow "Zawody",
           punkty_globalne
      from bd3_kluby k
     left join bd3_zawodnicy z on k.nr_klubu = z.nr_klubu
     left join bd3_wyniki w on w.nr_zawodnika = z.nr_zawodnika
)
pivot
(
    sum( punkty_globalne )
    for "Zawody" in
        ( 1 as "Zawody I", 2 as "Zawody II", 3 as "Zawody III", 4 as "Zawody IV" )
)
order by "Nr klubu";

```

Nr klubu	Klub	Zawody I	Zawody II	Zawody III	Zawody IV
1 Allianz Warszawa		341	268	395	421
2 KB Orientuz Warszawa		(null)	55	51	22
3 KB Gymnasium Warszawa		498	422	408	435
4 Legia Warszawa		77	51	75	76
5 Flota Gdynia		(null)	(null)	(null)	(null)
6 KB Promyk Ciechanów		153	120	158	137
7 KB Pułaski Strong Warka		(null)	(null)	(null)	(null)
8 AZS SGGW Warszawa		112	83	43	(null)
9 Grunwald Poznań		(null)	(null)	(null)	(null)
10 AZS Uniwersytet Warszawski		71	95	48	80
11 KB Amator Mińsk Mazowiecki		(null)	(null)	(null)	(null)
12 Śląsk Wrocław		(null)	(null)	(null)	(null)
13 KB Promyk Ursus		(null)	39	(null)	(null)
14 AZS-AWF Warszawa		41	61	33	25

...

Powyższe zdanie ma wadę polegającą na statycznej konstrukcji z uwagi na liczbę zawodów oraz dodatkowo na brak możliwości zrealizowania podsumowań według obu wymiarów (zawody i kluby).

7. Współczesne środowiska programowe wyspecjalizowane w tworzeniu raportów na podstawie danych z bazy (na przykład Jasper Reports) mają "zaszytą" w sobie powyższą metodę. W takim przypadku wystarczy opracować standardowe zdanie `select`, na podstawie którego zostanie wygenerowany raport krzyżowy, na przykład:

```
select z.nr_kategorii, nazwa_kategorii, w.nr_zawodow
from bd3_zawodnicy z
join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
join bd3_kategorie k on k.nr_kategorii = z.nr_kategorii;
```

Na jego podstawie jest możliwe wygenerowanie raportu (na przykład w formacie pdf) tego samego z przedstawionym powyżej i zawierającym podsumowanie według obu wymiarów.

Zadanie do samodzielnego wykonania:

1. Zdanie `select`:

```
select rezultat_min, rezultat_sek
from bd3_wyniki;
```

tworzy poniższy zbiór:

REZULTAT_MIN	REZULTAT_SEK
61	36
63	34
60	52
61	7
48	51
71	2

-- brak 0 prowadzącego dla wartości < 10

Należy opracować perspektywę `bd3_wyniki_zawodow` tworzącą zbiór o postaci:

NR_ZAWODOW	ZAWODNIK	KLUB	REZULTAT
3 Czechowski		Warszawianka	34:12
4 Karpisz		KB Trucht Warszawa	33:00
1 Setniewski		KB Lechici Zielonka	32:29
1 Gliniewicz		KB Gymnasium Warszawa	43:24
4 Cieślak		Warszawianka	33:32
2 Chmielowiec		KB Legionowo	49:41
3 Karpisz		KB Trucht Warszawa	34:23
4 Pałasz		KB Trucht Warszawa	41:06
3 Pałasz		KB Trucht Warszawa	44:47

, a następnie w oparciu o nią tworzyć zbiory wyników dla poszczególnych zawodów wykorzystując zmienną wiązania (`:Nr_zawodow`).

Należy wykorzystać konstrukcję `case` oraz funkcję `to_char` w połączeniu z działaniem konkatenacji.

Zapytania zagnieżdżone (podzapytania) - podstawy

Jednym ze sposobów optymalizacji zdań SQL jest odpowiednie budowanie ich składni. Składnia języka daje kontrolę nad kolejnością przeprowadzanych złączeń przez odpowiednie konstruowanie fraz *from* i *where*.¹

Drugim mechanizmem umożliwiającym otrzymanie identycznych rezultatów bez konieczności wykonywania złączeń (lub ich ograniczania) są zapytania zagnieżdżone. Stosowanie podzapytań dynamizuje zdanie SQL, gdyż w jednej konstrukcji można zawrzeć ciąg kilku zdań SQL w celu osiągnięcia tego samego wyniku.

Zapytanie zagnieżdżone to konstrukcja zawierająca pełne zdanie *select* umieszczone w innym zdaniu *select*. Rezultat wewnętrznego zdania *select* (podzapytania) staje się zbiorem wejściowym dla wyrażenia zewnętrznego.

Postać ogólna zdania zagnieżdżonego przedstawia się następująco:

```
select kolumna_1, kolumn_2,.....
  from tabela1
 where kolumna_N operator ( select kolumna_M
                               from tabela2
                               where ....... );
```

Obowiązują następujące zasady budowania zapytania zagnieżdżonego:

1. Wewnętrzne zdanie ujęte jest w nawiasy () i zawsze zaczyna się klauzulą **select**,
2. Wewnętrzne zdanie musi zwracać albo wielkość skalarną (pojedyncza liczba, napis lub data) lub wektorową czyli wartości z jednej kolumny,²
3. Operatorami mogą być operatory używane do tej pory we frazie *where* (=, <, >, itp), jak również operatory mnogościowe, takie jak: *in*, *any*, *all* (łączone lub nie z operatorem *not*).

Przykładowo zdanie:

```
select nazwisko, imie, to_char ( data_urodzenia, 'DD-MM-YYYY' ) "Data urodzenia"
  from bd3_zawodnicy
 where data_urodzenia = ( select min ( data_urodzenia )
                           from bd3_zawodnicy
                           where plec = 'M' )
   and plec = 'M';
```

nakazuje znaleźć dane najstarszego zawodnika (lub zawodników) płci męskiej:

NAZWISKO	IMIE	Data urodzenia
1 Werthein	Edmund	08-03-1926

Zdanie wewnętrzne zwraca konkretną wartość (jedną datę) i dlatego w zdaniu zewnętrznym można użyć operatora porównania (=).

¹ Optymalizacja zdań SQL nie będzie omawiana w tym materiale.

² Daje się zauważać zmianę w tym zakresie, np. w systemie Oracle podzapytanie może zwracać wartości z kilku kolumn.

Ale poniższe zdanie:

```
select nazwisko, imie
from bd3_zawodnicy
where nr_zawodnika = (
    select nr_zawodnika
    from bd3_wyniki
    where nr_zawodow = 2 );
```

nie może być zrealizowane:

Error report - ORA-01427: jednowierszowe podzapytanie zwraca więcej niż jeden wiersz

, gdyż zdanie wewnętrzne zwraca zbiór wartości, a nie jedną liczbę. W tym przypadku trzeba zastosować operator mnogościowy.

Istnieją dwa typy zapytań zagnieżdżonych: skorelowane i nieskorelowane.
W zapytaniu nieskoreowanym interpreter poleceń SQL potrafi zakończyć przetwarzanie wewnętrznego zdania `select` przed przystąpieniem do analizy zdania zewnętrznego.

W powyższym przykładzie dotyczącym znajdowania najstarszych zawodników płci męskiej w ewidencji kolejność wykonywania zdania jest następująca:

1. Wykonywane jest jednorazowo zdanie wewnętrzne, efektem czego jest konkretna wartość wskazująca na najstarszą datę urodzenia (1926/03/08),
2. Obliczona wielkość wstawiana jest do zdania zewnętrznego i powstaje warunek:

.....
`where data_urodzenia = ' 26/03/08 ';`

3. Realizowane jest zdanie zewnętrzne.

Z kolei w zapytaniu skoreowanym interpreter nie jest w stanie wykonać zapytania wewnętrznego bez informacji pochodzących ze zdania zewnętrznego.

Przykładowo zdanie:

```
select nr_zawodnika, nr_klubu
from bd3_zawodnicy z
where extract ( year from sysdate ) - extract ( year from data_urodzenia ) >
( select avg ( extract ( year from sysdate ) - extract ( year from data_urodzenia ) )
  from bd3_zawodnicy x
  where x.nr_klubu = z.nr_klubu );
```

realizuje zestawienie tych zawodników, których wiek jest większy od średniej wieku wszystkich zawodników **jego** klubu.

Kolejność wykonywania się tego zdania jest następująca:

1. Na podstawie zdania zewnętrznego pobierany jest pierwszy wiersz z tabeli BD3_ZAWODNICY o aliasie z,
2. Numer klubu z tego wiersza (`z.nr_klubu`) wstawiany jest do zdania wewnętrznego i na tej podstawie wyliczana jest średnia wieku zawodników jednego klubu,
3. Wynik ten zwracany jest do zdania zewnętrznego i na podstawie warunku we frazie `where` następuje, bądź nie, kwalifikacja pierwotnie odczytanego wiersza do zbioru wynikowego,

4. Czytany jest kolejny wiersz z tabeli BD3_ZAWODNICY z aliasem z i cykl się powtarza.

Zapytania skorelowane wymagają wielokrotnego wykonania wewnętrznego `select` i dlatego są mało wydajne.

Wadą tą nie są obciążone zapytania nieskorelowane, które można wykorzystywać do zastępowania złączeń i przyspieszania operacji na bazie danych, na przykład:

Zdanie:

```
select nr_zawodnika, z.nr_klubu
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
where nazwa_klubu like '%AZS Uniwersytet%';
```

można zastąpić zdaniem:

```
select nr_zawodnika, nr_klubu
from bd3_zawodnicy
where nr_klubu = ( select nr_klubu
                    from bd3_kluby
                    where nazwa_klubu like '%AZS Uniwersytet%' );
```

Operatory mnogościowe

W języku SQL stosuje się trzy operatory mnogościowe: `IN`, `ANY` i `ALL`, służące do budowy zapytań zagnieżdżonych w przypadku, gdy zapytanie wewnętrzne zwraca wiele wartości, na przykład zbiór numerów zawodników czy zbiór numerów klubów.

IN

Przykład:

Należy utworzyć listę zawodników, którzy brali udział w cyklu zawodów czyli wystąpili w co najmniej jednym biegu (analiza aktywności zawodników).

Poprzez równozłączenie zdanie to będzie wyglądało tak:

```
select distinct nazwisko, imie
from bd3_wyniki w join bd3_zawodnicy z
on w.nr_zawodnika = z.nr_zawodnika
order by nazwisko;
```

Równoważnym zdaniem skonstruowanym poprzez zapytanie zagnieżdżone i operator `IN` jest zdanie:

```
select nazwisko, imie
from bd3_zawodnicy
where nr_zawodnika in ( select nr_zawodnika
                           from bd3_wyniki )
order by nazwisko; -- zbiór zawiera 452 nazwiska
```

Zdanie wewnętrzne tworzy zbiór wynikowy w postaci kolumny `nr_zawodnika`, która zawiera wszystkie numery zawodników biorących udział w cyklu zawodów. Zdanie zewnętrzne dokonuje wyboru kolumn `nazwisko` i `imie` dla wybranych numerów. Warto zauważyć, że w powyższej konstrukcji zdanie wewnętrzne tworzy zbiór niepowtarzających się numerów zawodników (czyli stosowana jest niejawnie klauzula `distinct`).

Można również stosować operator *NOT IN* będący zaprzeczeniem operatora *IN*, na przykład w celu określenia liczby czy wyboru zawodników, którzy nie startowali w żadnych zawodach, a są w ewidencji:

```
select count(*)
from bd3_zawodnicy
where nr_zawodnika not in ( select nr_zawodnika
                                from bd3_wyniki );
-- zbiór zawiera 319 nazwisk
```

ANY i ALL

Podobnie jak *IN*, operator *ANY* (ang. any – jakikolwiek) oraz *ALL* (ang. all - wszystkie) umożliwia porównanie kolumny tabeli do zbioru wartości.

W swej najprostszej postaci *ANY* jest odpowiednikiem *IN*, gdyż:

IN jest równoważne zapisowi = ANY

Przykładowo zdanie:

```
select nr_klubu, nazwa_klubu
from bd3_kluby
where nr_klubu = any ( select nr_klubu
                            from bd3_zawodnicy
                            where nr_klubu > 15 )
order by nr_klubu;
```

tworzy zestawienie klubów o numerach powyżej 15, które mają zarejestrowanych zawodników.

Operator *ANY* i *ALL* może być używany łącznie z innymi operatorami relacji, takimi jak *>*, *<*, lub *<>*. Rozważmy przykładowe zdanie:

```
select A
      from Tabela_Zewnętrzna
     where A < any ( select B
                           from Tabela_Wewnętrzna );
```

Założymy, że zawartość obu tabel jest następująca:

Tabela_Zewnętrzna	Tabela_Wewnętrzna
2	3
4	5
6	7
8	
10	

Zbiór wynikowy powyższego zdania zawierał będzie wartości: 2, 4 i 6, gdyż dla każdej z nich można w Tabeli_Wewnętrznej dobrać wartość od niej większą. Innymi słowy warunek *where* tego zdania można czytać jako: „.....jeśli A jest mniejsze od jakiejkolwiek wartości zbioru B.”

Natomiast zdanie:

```
select A
      from Tabela_Zewnętrzna
     where A > any ( select B
                           from Tabela_Wewnętrzna );
```

zwróci wartości: 4, 6, 8 i 10, gdyż tylko 2 nie jest większe od jakiejkolwiek wartości zbioru B.
W przypadku użycia operatora *ALL* zdanie:

```
select A
  from Tabela_Zewnętrzna
 where A > all ( select B
      from Tabela_Wewnętrzna );
```

należy czytać jako: „.....jeśli A jest większe od wszystkich wartości zbioru B.”
Zbiór wynikowy będzie składał się z liczb 8 i 10, gdyż tylko one są większe od wszystkich wartości w zbiorze B.

Uwagi:

- Jeśli zbiór wartości zwrocony przez zdanie wewnętrzne jest zbiorem pustym, wtedy zdanie zewnętrzne też zwróci zbiór pusty.

Przykładowo zdanie:

```
select nazwisko, imie
  from bd3_zawodnicy
 where nr_zawodnika = ( select nr_zawodnika
      from bd3_wyniki
     where nr_zawodow = 5 );
```

zwróci zbiór pusty, gdyż brak jest w bazie wyników zawodów o numerze 5.

	NAZWISKO		IMIE
--	-----------------	--	-------------

- Konstrukcji zagnieżdżonych można używać również w zdaniach DML (*insert*, *update*, *delete*).

Przykładowo:

```
update bd3_zawodnicy
  set nr_klubu = ( select nr_klubu
      from bd3_zawodnicy
     where nr_zawodnika = 334 )
 where nr_zawodnika < 300
   and plec = 'M';
```

modyfikuje tabelę BD3_ZAWODNICY w ten sposób, że mężczyzn o numerach mniejszych niż 300 przenosi do klubu, w którym jest zarejestrowany zawodnik o numerze 334.

A zdanie:

```
delete bd3_zawodnicy
 where nr_zawodnika not in
 ( select w.nr_zawodnika
   from bd3_wyniki w join bd3_zawodnicy z on w.nr_zawodnika = z.nr_zawodnika
      join bd3_zawody za on w.nr_zawodow = za.nr_zawodow
    where extract( year from data_zawodow ) between extract( year from sysdate ) - 10
      and extract( year from sysdate )
 );
```

kasuje z ewidencji zawodników, którzy nie startowali w zadanym okresie w żadnych zawodach, mino, że mogli startować wcześniej.

Uwaga: Aby powyższe zdanie mogło się wykonać należy zmodyfikować model BiegBaza tak, aby możliwe było kasowanie obiektów nadzędnych wraz z pozostałymi z nimi w relacji obiektami podrzędnymi. W tym przypadku obiektem nadzędnym jest wiersz w tabeli BD3_ZAWODNICY, a obiektami podrzędnymi wiersze w tabeli BD3_WYNIKI jego dotyczące.

Można to uczynić zdaniami SQL modyfikującym relacje między tymi tabelami:

```
alter table bd3_wyniki
drop constraint fk_bd3_zawodnicy_bd3_wyniki;

alter table bd3_wyniki
add constraint fk_bd3_zawodnicy_bd3_wyniki foreign key (nr_zawodnika)
references bd3_zawodnicy (nr_zawodnika) on delete cascade;
```

3. Nie istnieją żadne ścisłe ograniczenia związane z głębokością zagnieżdżania zdań wewnętrznych³:

```
select .....
from tabela_1
where kolumna_1 in ( select kolumna_2 from tabela_2
                      where kolumna_3 = ( select .....
                                           where kolumna_4 in ..... ));
```

, jak również liczbą zdań wewnętrznych zastosowanych jednocześnie, na przykład:

```
select * from bd3_zawodnicy
where nr_klubu = ( select nr_klubu
                     from bd3_kluby
                     where nazwa_klubu like '%Gymnasion%' )
and nr_kategorii in ( select nr_kategorii
                       from bd3_kategorie
                       where nazwa_kategorii in ( 'II', 'III', 'IV' ) )
order by extract ( year from data_urodzenia ), nazwisko, imie;
```

4. Podsumowując zagadnienie zapytań zagnieżdżonych należy pamiętać, że im większe są tabele bazodanowe, tym wydajniejsza jest ta metoda w porównaniu z metodą łączenia tabel.

Zapytania zagnieżdżone - zaawansowane zastosowania

Do tej pory omawiane było zastosowanie zapytań zagnieżdżonych do celów filtrowania w zdaniu `select`, `update` lub `delete` czyli były one umieszczane we frazie `where.....` oraz do modyfikacji danych w tabeli czyli w zdaniu `update` we frazie `set....`.

Technika zapytań zagnieżdżonych może być stosowana również w innych frazach zdania `select` takich jak fraza `select...`, `from...` i `having...`, jak również w zdaniu `insert` we frazie `values ...`.

Podzapytanie we frazie `having` zdania `select`

Fraza `having` powoduje, że zbiór wyników generowanych przez zdanie `select` z funkcją agregującą zawiera tylko te wiersze, które spełniają warunek logiczny zawarty we frazie `having`.

Na przykład zdanie:

```
select nazwa_klubu "Klub", count ( * ) "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
group by nazwa_klubu
order by "Liczba zawodników" desc;
```

³ Oracle pozwala na używanie zagnieżdżeń na 255 poziomach we frazie `where`, co wydaje się być wielkością czysto teoretyczną.

utworzy zbiór:

Klub	Liczba zawodników
Allianz Warszawa	146
KB Gymnasion Warszawa	104
KB Trucht Warszawa	100
KB Lechici Zielonka	73
AZS Uniwersytet Warszawski	72
KB Promyk Ciechanów	46
AZS SGGW Warszawa	39
Warszawianka	32
KB Legionowo	32

Chcąc ograniczyć liczbę wierszy w tym zbiorze należy zastosować frazę *having*:

```
select nazwa_klubu "Klub", count( * ) "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
group by nazwa_klubu
having count( * ) >= 100
order by "Liczba zawodników" desc;
```

i wtedy zbiór wyników zostanie zawężony:

Klub	Liczba zawodników
Allianz Warszawa	146
KB Gymnasion Warszawa	104
KB Trucht Warszawa	100

Użycie warunku ograniczającego *count(*) >= 100* we frazie *where* powoduje błąd wykonania:

```
SQL Error: ORA-00934: funkcja grupowa nie jest tutaj dozwolona
00934. 00000 -  "group function is not allowed here"
```

Można natomiast użyć frazy *where* do filtrowania wierszy przed użyciem funkcji agregującej, na przykład:

```
select nazwa_klubu "Klub", count( * ) "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
where plec = 'K'
group by nazwa_klubu
having count( * ) > 10
order by "Liczba zawodników" desc;
```

Klub	Liczba zawodników
KB Gymnasion Warszawa	38
Allianz Warszawa	18
KB Trucht Warszawa	13

Fraza *where plec = 'K'* filtryuje wiersze z tabeli BD3_ZAWODNICY dotyczące tylko kobiet i tylko te wiersze są poddane funkcji agregującej *count*. Natomiast fraza *having*, w powstałym zbiorze wynikowym, pozostawia tylko te wiersze, które spełniają warunek użyty w niej.

W obu zaprezentowanych przykładach z frazą *having* można zauważać, że w zdaniu tam występującym jedna ze stron tego zdania zawiera konkretną wartość (100 lub 10). W takim razie należy przypuszczać, że zamiast konkretnej wartości można zastosować podzapytanie wzorem poprzednio omawianych zagadnień.

Na przykład zdanie:

```
select nazwa_klubu "Klub", count( * ) "Liczba zawodników"
from bd3_zawodnicy z
join bd3_kluby k on z.nr_klubu = k.nr_klubu
where plec = 'K'
group by nazwa_klubu
having count( * ) >= ( select count( * )
                           from bd3_zawodnicy
                           where nr_klubu = 31 and plec = 'K' )
order by "Liczba zawodników" desc;
```

wygeneruje dokładnie ten sam zbiór wynikowy, co poprzednie:

Klub	Liczba zawodników
KB Gymnasium Warszawa	38
Allianz Warszawa	18
KB Trucht Warszawa	13

Istotna różnica jest taka, że poprzednio "na sztywno" ustawiona była wartość 10, a teraz ten warunek ma charakter dynamiczny, gdyż liczba zarejestrowanych kobiet w klubie o numerze 31 może ulegać zmianie z upływem czasu. Przy pomocy tak opracowanego zdania *select* reguła biznesowa mówiąca: "...generuj zawsze raport o klubach, w których liczba kobiet jest nie mniejsza od liczby kobiet w klubie KB Trucht Warszawa..." będzie zawsze spełniona.

Drugim przykładem jest zdanie:

```
select nazwa_klubu "Klub",
       round( avg( extract( year from sysdate ) - extract( year from data_urodzenia ) ) ) "Średnia wieku"
  from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
 where plec = 'K'
 group by nazwa_klubu
 having avg( extract( year from sysdate ) - extract( year from data_urodzenia ) ) < 40
 order by "Średnia wieku" desc;
```

tworzące zbiór wynikowy:

Klub	Średnia wieku
Bielanski KB Warszawa	38
AZS-AWF Warszawa	36

czyli zestawienie klubów, w których średnia wieku kobiet nie przekracza 40 lat.

Realizacja tego zdania odbywa się według scenariusza:

1. Wybierane są z tabeli BD3_ZAWODNICY wiersze dotyczące kobiet i łączone z odpowiadającymi im nazwami klubów z tabeli BD3_KLUBY,
2. Obliczane są średnie wieku kobiet dla poszczególnych klubów,
3. Z tak otrzymanego zbioru filtrowane są tylko te wiersze, w których ta średnia jest mniejsza od 40 lat.

Warto zauważyć, że we frazie *having*, jak również *group by* nie można używać aliasów.

Trzeci przykład: Chcemy utworzyć podobny do poprzedniego zbiór wynikowy prezentujący tylko te kluby, w których średnia wieku kobiet jest mniejsza od średniej wieku wszystkich zaewidencjonowanych kobiet.

Zdanie pomocnicze:

```
select avg ( extract ( year from sysdate ) - extract ( year from data_urodzenia ))
from bd3_zawodnicy
where plec = 'K';
```

oblicza tę średnią:

AVG(EXTRACT(YEARFROMSYSDATE)-EXTRACT(YEARFROMDATA_URODZENIA))
45,38461538461538461538461538461538461538461538

Zatem zdanie zasadnicze można skonstruować tak:

```
select nazwa_klubu "Klub",
       round ( avg ( extract ( year from sysdate ) - extract ( year from data_urodzenia )) ) "Średnia wieku"
  from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
 where plec = 'K'
 group by nazwa_klubu
 having avg ( extract ( year from sysdate ) - extract ( year from data_urodzenia )) <
        ( select avg ( extract ( year from sysdate ) - extract ( year from data_urodzenia ))
          from bd3_zawodnicy
         where plec = 'K' )
 order by "Średnia wieku" desc;
```

i otrzymamy zbiór:

Klub	Średnia wieku
AZS Uniwersytet Warszawski	45
KB Gymnasjon Warszawa	45
KB Trucht Warszawa	44
KB Lotos Jabłonna	44
Warszawianka	43
Legia Warszawa	43
KB Promyk Ciechanów	41
Bielanski KB Warszawa	38
AZS-AWF Warszawa	36

Uwaga:

1. Należy zawsze pamiętać, że zdanie podzielone musi zaczynać się frazą `select` i być umieszczone w nawiasach okrągłych.
2. Zaokrąglenie liczb zostało zastosowane tylko do wyświetlenia końcowych wyników (we frazie `select zdania zewnętrznego`), a nie do liczenia średnich i filtrowania we frazie `having`.

Podzapytanie we frazie `from zdania select`

Fraza `from` w swej wersji podstawowej zawiera nazwy tabel, z których pochodzą kolumny użyte do tworzenia zbioru wynikowego. W wersji zaawansowanej dopuszcza się, aby na liście tej frazy znajdowały się dowolne zbiory, a więc oprócz tabel także podzapytania w postaci zdań `select`. Przykładem niech będzie zdanie:

```
select nazwisko || ' ' || imie as "Zawodnik", nazwa_klubu as "Klub"
  from bd3_zawodnicy z, ( select nazwa_klubu, nr_klubu
                           from bd3_kluby
                          where nr_klubu in ( 31, 10) ) k
```

```
where z.nr_klubu = k.nr_klubu
and plec = 'K'
order by "Zawodnik";
```

Zasady łączenia tabel omawiane wcześniej mają w powyższej konstrukcji zastosowanie. Zdanie `select` o aliasie `k` ogranicza zbiór klubów do dwóch i ten zbiór jest łączony z tabelą BD3_ZAWODNICY poprzez `nr_klubu`.

Zawodnik	Klub
Antropik Jolanta	KB Trucht Warszawa
Biała Iza	AZS Uniwersytet Warszawski
Bobrownicka Sylwia	AZS Uniwersytet Warszawski
Dobosz Aleksandra	AZS Uniwersytet Warszawski
Jabłońska Anna	KB Trucht Warszawa

...

Oczywiście ten sam efekt można uzyskać łącząc ze sobą bezpośrednio dwie tabele BD3_ZAWODNICY i BD3_KLUBY. Ale w wielu przypadkach to rozwiązywanie może okazać się wydajniejsze (szczególnie w hurtowniach danych, gdy liczba wierszy w tabelach jest bardzo duża), gdyż jeden ze zbiorów wejściowych już na początku jest ograniczony tylko do dwóch wierszy.

Drugim praktycznym zastosowaniem podzapytań we frazie `from` jest problem limitowania liczby wierszy w zbiorze wynikowym.

Jak wygenerować zbiór zawierający sumaryczne wyniki dziesięciu najlepszych klubów lub opracować raport zawierający pięciu najlepszych zawodników w każdych zawodach?

Pseudokolumna `rownum`

Pseudokolumna (wirtualna) `rownum` numeruje wiersze w zbiorze wynikowym podobnie jak popularna kolumna Lp na papierowych zestawieniach.

Sposób użycia przedstawia poniższe zdanie:

```
select rownum, nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
from bd3_zawodnicy
where nr_klubu = 3;
```

i wygenerowany zbiór wynikowy:

ROWNUM	Zawodnik	Data urodzenia
1	Cieślak Petter	80/08/21
2	Gaca Wojciech	60/01/30
3	Bieniecki Piotr	80/06/03
4	Pardela Gerard	60/07/29
5	Sobczak Marcin	84/05/11
6	Krawczyński Dariusz	56/11/14
7	Przybysz Dariusz	78/12/08

...

Możliwe jest użycie pseudokolumny `rownum` do ograniczenia liczby wyświetlanych wierszy, na przykład:

```
select rownum, nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
from bd3_zawodnicy
where nr_klubu = 2 and rownum <= 5;
```

ROWNUM	Zawodnik	Data urodzenia
1	Cieślak Petter	80/08/21
2	Gaca Wojciech	60/01/30
3	Bieniecki Piotr	80/06/03
4	Pardela Gerard	60/07/29
5	Sobczak Marcin	84/05/11

Powyższy zbiór zawiera pięć wierszy, zgodnie z dyspozycją dotyczącą *rownum*, ale należy zadać sobie pytania: Co to są za wiersze? Według jakiego kryterium został dobrany ten zbiór wynikowy? Widać, że nie jest to ani nazwisko ani data urodzenia zawodnika. Wybór jest przypadkowy i zależy od położenia wierszy w tabeli czyli w jakiej kolejności zostały do niej wprowadzane, na przykład zdaniem *insert*.

Po uwzględnieniu sortowania według, na przykład, nazwiska, czyli:

```
select rownum, nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
from bd3_zawodnicy
where nr_klubu = 3 and rownum <= 5
order by nazwisko;
```

otrzymamy zbiór o zawartości:

ROWNUM	Zawodnik	Data urodzenia
3	Bieniecki Piotr	80/06/03
1	Cieślak Petter	80/08/21
2	Gaca Wojciech	60/01/30
4	Pardela Gerard	60/07/29
5	Sobczak Marcin	84/05/11

Zbiór jest posortowany według nazwisk, ale "liczba porządkowa" generowana przez *rownum* jest zakłócona. Po analizie tego przykładu można dojść do wniosku, że w czasie wykonywania się powyższego zdania *select* najpierw dokonuje się numeracja wierszy (*rownum*), a potem sortowanie zbioru wynikowego. I tak jest w istocie. Sortowanie zbioru wynikowego jest zawsze ostatnią czynnością wykonywaną w zdaniu *select*.

Limitowanie typu *rownum* <= wartość_progowa

W takim przypadku należy tak skonstruować zdanie *select*, aby najpierw odbyło się sortowanie, a potem numerowanie wierszy. Z pomocą przychodzi zastosowanie podzapytania we frazie *from*.

Zdanie:

```
select rownum, "Zawodnik", "Data urodzenia"
from (
    select nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
    from bd3_zawodnicy
    where nr_klubu = 3
    order by nazwisko
)
where rownum <= 5;
```

zostanie wykonane według scenariusza:

1. Wybranie danych o zawodnikach klubu o numerze 3 (zdanie wewnętrzne),
2. Posortowanie wybranych zawodników według nazwisk (zdanie wewnętrzne),
3. Ponumerowanie wierszy (*rownum* we frazie *select* zdania zewnętrznego),
4. Filtrowanie wierszy według frazy *where* zdania zewnętrznego.

Ostateczna postać zbioru wynikowego przedstawia się następująco:

ROWNUM	Zawodnik	Data urodzenia
1	Bałaszow Marcin	60/01/27
2	Banaszek Ewa	84/12/03
3	Banczyk Agnieszka	66/07/30
4	Bieda Marzena	56/05/15
5	Bielńska Krystyna	75/08/22

Zbiór zawiera pięć pierwszych nazwisk zawodników klubu o numerze 3.

Zmieniając klucz sortowania w podzapytaniu z nazwiska na datę urodzenia otrzymamy zbiór:

ROWNUM	Zawodnik	Data urodzenia
1	Werthein Edmund	26/03/08
2	Gradus Bożena	27/12/12
3	Bielowski Wojciech	31/07/23
4	Żelazko Jarosław	33/08/27
5	Cieślik Robert	41/08/06

prezentujący dane pięciu najstarszych zawodników tego klubu.

Limitowanie typu wartość_progowa_min <= rownum <= wartość_progowa_max

W analogiczny sposób należy postępować w przypadku, gdy limitowanie liczby wierszy dotyczy warunku:

.....rownum between numer_wiersza_od and numer_wiersza_do

Zdanie:

```
--3  select * from
  (
--2    select rownum "Lp", "Zawodnik", "Punktacja"
      from (
--1      select nazwisko "Zawodnik", sum ( nvl( punkty_globalne, 0 ) ) "Punktacja"
          from bd3_wyniki w join bd3_zawodnicy z on z.nr_zawodnika = w.nr_zawodnika
          group by nazwisko
          order by "Punktacja" desc
      )
  )
 where "Lp" between 11 and 15;
```

wygeneruje zbiór:

Lp	Zawodnik	Punktacja
11	Osińska	155
12	Gliwińska	143
13	Piotrowska	142
14	Ryggen	141
15	Sitarek	136

Zdanie to zostanie wykonane według scenariusza:

1. Wybranie danych zgodnie ze zdaniem --1 (posortowanie malejąco według punktacji),
2. Ponumerowanie wierszy od 1 i nadanie tej kolumnie aliasu "Lp" (select --2),
3. Filtrowanie zbioru według "Lp" w zakresie [11..15] (select --3).

Są to wiersze zbioru od pozycji 11 do 15 według kryterium kolumny "Punktacja".

Warto zauważyć, że w zdaniu `select --2` kolumna "Punktacja" nie jest już agregatem tylko kolumną skalarną, co oznacza, że w tym zdaniu można by było użyć filtrowania ...`where "Punktacja" > 100...`, a nie ...`having "Punktacja" > 100...`.

Pseudokolumna `rownum` została opatrzona aliasem "Lp", dzięki czemu w zdaniu `select --3` nie posiada już właściwości `rownum`, tylko jest zwykłą kolumną, której można używać do filtrowania wierszy.

Limitowanie typu `rownum >= wartość_progowa`

Limitowanie według `rownum` oznacza sprecyzowanie, ile wierszy ma znaleźć się w zbiorze wynikowym. Warunek `rownum <= 5` determinuje co najwyżej pięć wierszy, warunek `rownum between 11 and 15` - również.

Na podstawie warunku `rownum >= 5` nie można określić górnej granicy liczby wierszy. Dlatego zdanie z jednego z poprzednich przykładów ze zmodyfikowanym warunkiem filtrowania:

```
select rownum, "Zawodnik", "Data urodzenia"
from ( select nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
       from bd3_zawodnicy
       where nr_klubu = 3
       order by nazwisko )
      where rownum >= 5;
```

zwraca zbiór pusty.

Ponieważ limitowanie zawsze musi być związane z określeniem jakiegoś kryterium należy na etapie sortowania dokonać odpowiedniego uporządkowania zbioru. Zdanie powyższe może wyglądać tak:

```
select rownum, "Zawodnik", "Data urodzenia"
from ( select nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
       from bd3_zawodnicy
       where nr_klubu = 3
       order by nazwisko desc )
      where rownum <= 5;
```

a wynik:

ROWNUM	Zawodnik	Data urodzenia
1	Żelazko Jarosław	33/08/27
2	Zieleńska Elżbieta	75/03/30
3	Zdybel Marcin	69/01/30
4	Wyszyński Szymon	50/01/21
5	Wróblewski Dariusz	53/11/23

Został odwrócony kierunek sortowania czyli w zbiorze wynikowym znalazło się pięć ostatnich nazwisk, ale uporządkowanych począwszy od ostatniego.

Jeśli istnieje potrzeba posortowania otrzymanego zbioru według nazwisk w porządku naturalnym należy zagnieździć opracowane zdanie:

```
--3    select rownum "Lp", "Zawodnik", "Data urodzenia"
      from
--2      ( select "Zawodnik", "Data urodzenia
            from
--1          ( select "Zawodnik", "Data urodzenia"
```

```

from ( select nazwisko || ' ' || imie "Zawodnik", data_urodzenia "Data urodzenia"
      from bd3_zawodnicy
      where nr_klubu = 3
      order by nazwisko desc )
      where rownum <= 5
    )
  order by "Zawodnik"
);

```

Zbiór wynikowy tworzony jest według scenariusza:

1. Na podstawie `select --1` powstaje zbiór pięciu wierszy z nazwiskami z końca alfabetu posortowany odwrotnie czyli od Ż do W,
2. Zbiór ten jest sortowany według naturalnego porządku nazwisk zdaniem `select --2` czyli od W do Ż,
3. Powstały zbiór jest numerowany w zdaniu `select --3`.

Uwaga:

Począwszy od wersji Oracle 12c zdanie `select` zostało wzbogacone o implementację limitowania bez potrzeby używania konstrukcji przedstawionych powyżej. Szczegóły zawarte są w dokumentacji Oracle lub w innych zasobach internetowych pod hasłem (*row limiting clause oracle*).

Podzapytanie we frazie select zdania select

W podstawowej swojej konstrukcji zdanie `select` zawiera we frazie `select` nazwy kolumn lub funkcje agregujące opatrzone lub nie aliasami.

Istnieje możliwość umieszczenia w tej frazie podzapytania zwracającego wartość skalarną.

Przy pomocy zdania:

```

select nazwisko||' '||imie "Zawodnik", nazwa_klubu "Klub", sum(punkty_globalne) "Pkt indywidualne"
  from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
  join bd3_wyniki w on z.nr_zawodnika = w.nr_zawodnika
group by nazwisko, imie, nazwa_klubu
having sum(punkty_globalne) is not null
order by "Pkt indywidualne" desc;

```

tworzony jest standardowy zbiór wynikowy zawierający osiągnięcia zawodników w postaci sumarycznej liczby punktów przez nich zdobytych:

Zawodnik	Klub	Pkt indywidualne
Muzyka Urszula	Legia Warszawa	198
Karpisz Zbigniew	KB Trucht Warszawa	194
Pałasz Joanna	KB Trucht Warszawa	192
Ciemski Jan	KB Lechici Zielonka	189
Gawryszewski Radosław	KB Trucht Warszawa	187
Margoła Bartłomiej	Allianz Warszawa	185

Chcemy poszerzyć ten zbiór o dodatkowe kolumny zawierające sumaryczną ilość punktów zdobytych przez wszystkich zawodników klubu, do którego należy zawodnik prezentowany w każdym wierszu zbioru oraz liczbę zawodników należących do tego samego klubu.

Zdanie może wyglądać tak:

```

select nazwisko "Zawodnik", nazwa_klubu "Klub"
,( select suma_z from ( select sum ( nvl(punkty_globalne, 0 ) ) suma_z
                           from bd3_wyniki wz
                           where wz.nr_zawodnika = z.nr_zawodnika )
) "Pkt zawodnika"
,( select suma_k from ( select sum ( nvl(punkty_globalne, 0 ) ) suma_k
                           from bd3_wyniki wk, bd3_zawodnicy zk
                           where zk.nr_klubu = z.nr_klubu and zk.nr_zawodnika = wk.nr_zawodnika )
) "Pkt klubu"
,( select liczba_z from ( select count(*) liczba_z
                           from bd3_zawodnicy zl
                           where z.nr_klubu = zl.nr_klubu )
) "Liczność klubu"
from bd3_zawodnicy z, bd3_kluby k
where z.nr_klubu = k.nr_klubu
order by "Zawodnik";

```

, a fragment zbioru wynikowego:

Nr_zawodnika	Zawodnik	Klub	Pkt zawodnika	Pkt klubu	Liczność klubu
787 Chrapek	Warszawianka		(null)	218	32
855 Chruściński	Warszawianka		0	218	32
684 Chrzanowski	Allianz Warszawa		0	1425	146
657 Chrześciążański	KB Pułaski Strong Warka		(null)	0	8
460 Chudek	KB Lotos Jabłonna		28	432	17
547 Ciecierski	KB Promyk Ursus		(null)	39	16
831 Ciemski	KB Lechici Zielonka		(null)	821	73
840 Ciemski	KB Lechici Zielonka		189	821	73
303 Ciesielski	KB Promyk Ciechanów		63	568	46
268 Cieślak	KB Gymnasion Warszawa		0	1763	104
264 Cieślak	Warszawianka		97	218	32

Scenariusz wykonania tego zdania jest następujący:

1. Zdaniem zewnętrznym `select` pobierane są dane o pierwszym zawodniku ze zbioru uporządkowanego według alfabetu,
2. Z odczytanego wiersza pobierany jest numer zawodnika `z.nr_zawodnika` i "przenoszony" do podzapytania `select` w celu zsumowania jego wyników,
3. Z tego samego wiersza odczytywany jest numer klubu, do którego zawodnik należy `z.nr_klubu` i na jego podstawie w podzapytaniu `select` obliczane są sumaryczne punkty tego klubu (we frazie `where` jest ustawiony filtr `zk.nr_klubu = z.nr_klubu`).
4. Z tym samym numerem klubu `z.nr_klubu` realizowane jest trzecie podzapytanie `select` obliczające liczbę zawodników w tym klubie.
5. Zdaniem zewnętrznym `select` pobierane są dane kolejnego zawodnika.

Uwagi:

1. Na liście `select` znajdują się trzy podzapytania skorelowane ze zdaniem głównym.
2. Każde z tych podzapytań musi zwracać wielkość skalarną, stąd też każde z nich jest skonstruowane w sposób zapewniający:

`select skalar from (select agregat from...)`

3. W konstrukcji występują fizycznie te same zbiory danych, ale opatrzone różnymi aliasami czyli logicznie są to różne zbiory, na przykład tabela BD3_ZAWODNICY występuje jako alias **z** w zdaniu zewnętrznym oraz jako alias **zk** i **zl** w podzapytaniach.
4. W zaprezentowanym zbiorze wyników w kolumnie "Pkt zawodnika" występują wartości 0 oraz *null*. Wartość 0 oznacza, że dany zawodnik startował w zawodach, ale nie zdobył punktów w klasyfikacji generalnej czyli w tabeli BD3_WYNIKI figuruje. Wystąpienie *null* w tej kolumnie oznacza, że brak jest wyników zawodnika w tabeli wyników czyli zawodnik ten nie startował wcale. Zastosowanie funkcji *nvl* w podzapytaniu **select** umożliwia rozróżnienie tych faktów.

Wykonując zdanie:

```
select '787 Chrapek' "Zawodnik" , count( * ) "Liczba startów"
from bd3_wyniki
where nr_zawodnika = 787
union
select '855 Chruściński', count( * )
from bd3_wyniki
where nr_zawodnika = 855;
```

otrzymujemy wynik:

Zawodnik	Liczba startów
787 Chrapek	0
855 Chruściński	1

, co potwierdza powyższą tezę.

Zadania do samodzielnego wykonania:

1. Opracować zestawienie pokazujące zawodników i osiągnięte przez nich rezultaty, którzy należą do tego samego klubu, co zwycięzca zawodów nr 2 wśród mężczyzn.⁴
2. Opracować zestawienie kobiet należących do klubów, w których liczność zawodników przekracza liczbę 5.
3. Opracować zestawienie (Imię i nazwisko, Nazwa klubu i Nazwa Kategorii) pokazujące zawodniczki, które należą do klubów niewarszawskich i ich wiek jest większy od średniej wieku wszystkich kobiet w ewidencji.⁵
4. Opracować zdanie pokazujące zawodnika i zawodniczkę (Nazwisko i imię, przynależność klubowa oraz kategoria wiekowa), którzy zdobyli najwięcej punktów w klasyfikacji generalnej.
5. (trudne) Opracować zdanie pokazujące najlepszych zawodników i zawodniczek (Nazwisko i imię, przynależność klubowa oraz kategoria wiekowa), którzy zdobyli najwięcej punktów w klasyfikacji w kategoriach.

W tym zadaniu należy wykorzystać następujące techniki:

- zamiana minut i sekund na sekundy,
- tworzenie na liście *from* perspektyw chwilowych: pierwszej zawierającej najlepszy wynik w każdej kategorii wiekowej i drugiej znajdującej numer zawodnika w każdej kategorii wiekowej, który ten najlepszy wynik uzyskał.

⁴ Należy skorzystać ze zdania podrzędnego, a rezultaty wyświetlić w postaci mm:ss.

⁵ Należy wykonać raport przy pomocy zdania podrzędnego. Jako kryterium kwalifikacji do klubów warszawskich przyjąć nazwę klubu, w której znajduje się fragment słowa Warszawa (np. Warsz) w dowolnym miejscu nazwy.

	<i>Bazy Danych laboratorium</i>	Laboratorium BD7
--	-------------------------------------	---------------------

Zagadnienie: Zapoznanie się z SQL Developer Data Modeler w celu tworzenia logicznego i relacyjnego modelu bazy danych

SQL Developer Data Modeler jest narzędziem Oracle umożliwiającym tworzenie schematów logicznych, relacyjnych i generowanie skryptów umożliwiających implementację opracowanych modeli na różne wersje serwerów Oracle i nie tylko.

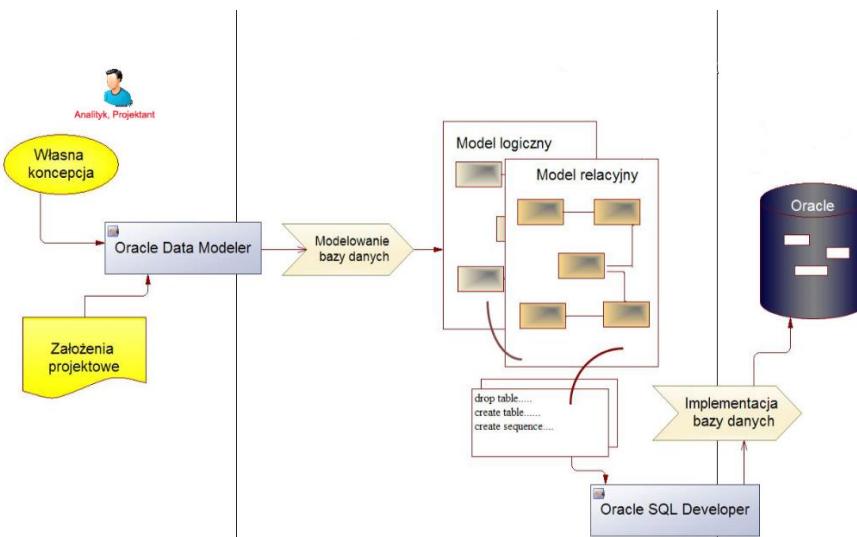
Oprogramowanie oraz dokumentacja znajdują się na stronie:

<https://www.oracle.com/database/technologies/appdev/datamodeler.html>

I. Etapy modelowania bazy danych

Modelowanie bazy danych składa się z kilku etapów:

1. Modelowanie pewnego fragmentu rzeczywistości czyli przedstawienie tej rzeczywistości w postaci modelu logicznego składającego się z encji i związków między encjami. Każda encja posiada swoją nazwę oraz strukturę, na którą składają się atrybuty opisujące dany obiekt i typy tych atrybutów. Co najmniej jeden z atrybutów musi być wyróżnikiem, przy pomocy którego można jednoznacznie identyfikować dany egzemplarz encji.
2. Na podstawie modelu logicznego tworzony jest model relacyjny, w którym na podstawie encji powstają tabele, a związki zostają transformowane na relacje. Tabele składają się z kolumn i ich typów. Pojawiają się klucze główne i klucze obce.
3. Model relacyjny poddawany jest analizie i ewentualnym modyfikacjom zmierzającym do przekształcenia go do wymaganych postaci normalnych (najczęściej do 3NF).
4. Na podstawie znormalizowanego modelu relacyjnego tworzony jest model fizyczny w języku SQL umożliwiający implementację na konkretnym serwerze bazodanowym.

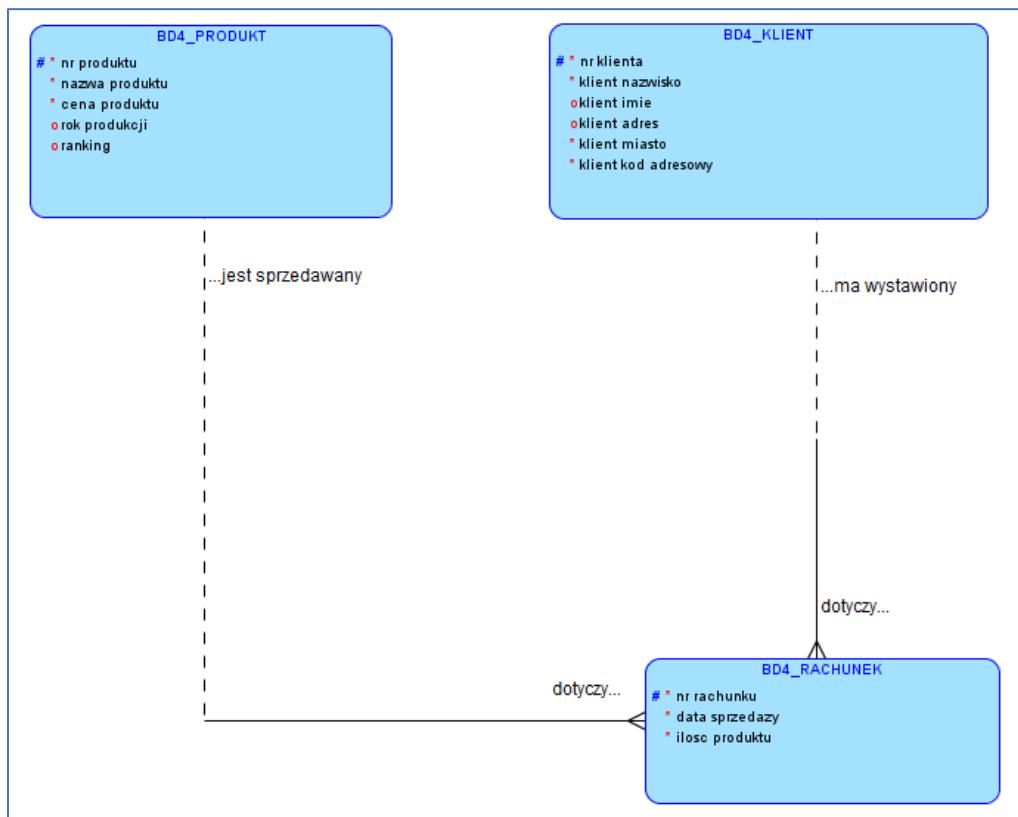


II. Projektowanie modelu bazodanowego z dwiema relacjami 1:N (one to many)

Należy opracować model logiczny *Rachunek* w oparciu o poniższe założenia.

Firma sprzedaje swoje produkty zarejestrowanym klientom i wystawia za te transakcje rachunki. Każda transakcja dotyczy jednego produktu czyli na rachunku widnieje tylko jedna pozycja sprzedanego towaru. W modelu *Rachunek* należy przechowywać informacje dotyczące sprzedaży produktów klientom według modelu: Klient, Produkt, Rachunek.

Finalna postać modelu logicznego przedstawiona jest na poniższym rysunku:



1. Zdefiniować trzy obiekty logiczne (*entity*) według poniższych założeń:

Entity BD4_PRODUKT:

Nazwa atrybutu	Typ atrybutu	Informacje dodatkowe
nr produktu	numeric (5)	Unikalny identyfikator
nazwa produktu	varchar (20)	Musi być wypełnione
cena produktu	numeric (8,2) ¹	Musi być wypełnione
rok produkcji	numeric (4)	
ranking	numeric (2)	Ranking produktów: ² 1 – produkt bardzo zły, 10 - produkt bardzo dobry

¹ Numeric(8,2) oznacza liczbę zawierającą 6 cyfr przed kropką dziesiętną i 2 cyfry po kropce. Należy ustawić Precision na 8 i Scale na 2.

² Znaczenie tego atrybutu należy opisać w komentarzu do definicji *entity* (Comments in RDBMS).

Entity BD4_KLIENT:

Nazwa atrybutu	Typ atrybutu	Informacje dodatkowe
nr klienta	numeric (4)	Unikalny identyfikator
klient nazwisko	varchar (30)	Musi być wypełnione
klient imie	varchar (15)	
klient adres	varchar (50)	
klient miasto	varchar (20)	Musi być wypełnione
klient kod adresowy	varchar (6)	Musi być wypełnione

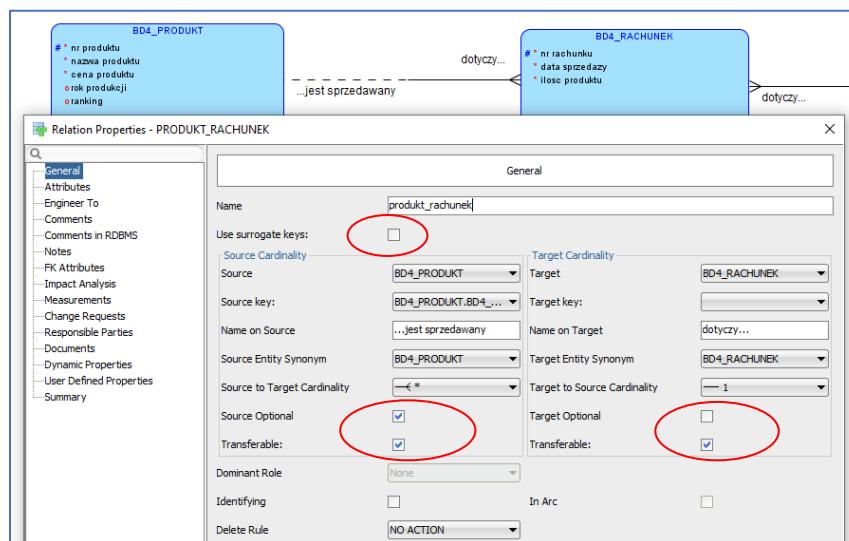
Entity BD4_RACHUNEK:

Nazwa atrybutu	Typ atrybutu	Informacje dodatkowe
nr rachunku	numeric (4)	Unikalny identyfikator
data sprzedaży	date	Musi być wypełnione
ilosc produktu	numeric (3)	Musi być wypełnione

2. Zapisać powstały fragment modelu logicznego pod nazwą *Rachunek* (save as...), a następnie poprzez funkcję *View/Logical Diagram Notation* zapoznać się z dwiema oferowanymi przez *Data Modeler* konwencjami prezentacji diagramów logicznych (notacje Barkera i Bachmana).
3. Zdefiniować związki między obiektami przyjmując następujące założenia:
 - Związek *produkt_rachunek* (jeden_do_wielu): każdy produkt może być wiele razy sprzedawany klientom, ale również mogą być produkty ani razu nie sprzedane.

W tym celu z paska narzędzi wybrać odpowiednią ikonę relacji 1:N, a następnie połączyć encje *BD4_PRODUKT* i *BD4_RACHUNEK* przeciągając myszką od encji *BD4_PRODUKT*.

Pojawi się formularz właściwości relacji, w którym można dokonać niezbędnych modyfikacji poprzez zmianę nazwy relacji, słownych interpretacji relacji (Name of Source i Name of Target)³ oraz siły relacji (Cardinality), np.:



Pozostawienie niewybranej opcji *Target Optional* oznacza, że w transakcji sprzedaży musi być określony produkt czyli w strukturze przyszłej tabeli *BD4_RACHUNEK* pozycja *nr_produktu*

³ Aby te określenia pojawiły się na diagramie należy na pulpicie głównym przy pomocy PPM (prawy przycisk myszy) wybrać opcję *Show / Labels*.

musi być określona. Gdyby ta opcja była zaznaczona oznaczałoby to, że kolumna nr_produktu będzie miała właściwość null czyli transakcja może dotyczyć nieokreślonego produktu.

- Związek *klient_rachunek* (jeden_do_wielu): każdy fakt sprzedaży dotyczy jednego określonego klienta, klient może wiele razy kupować produkty, ale są klienci, którzy nie dokonali żadnej transakcji.

Dla tego związku ustawić w taki sam sposób jego parametry jak dla klient_rachunek.

W panelu Brower odnaleźć definicje związków między encjami (Relations) i zapoznać się ze strategiami postępowania w przypadku kasowania wpisów (Delete Rule). Dla związku klient_rachunek ustawić strategię na restrykcyjną (RESTRICT), co oznacza, że nie będzie można usunąć wiersza z tabeli nadzędnej (BD4_KLIENT), jeśli w tabeli podrzędnej (BD4_RACHUNEK) znajdują się odpowiadające mu wiersze. Dla relacji produkt_rachunek ustawić strategię na CASCADE, co oznacza, że w przypadku kasowania wiersza w tabeli nadzędnej (BD4_PRODUKT), w tabeli podrzędnej (BD4_RACHUNEK) zostaną skasowane automatycznie wszystkie wiersze będące w relacji z kasowanym wierszem nadzędznym.⁴

4. Przy pomocy funkcji *Engineer To Relational Model* (PPM na poziomie *Logical Model* lub ikona **>>**) wygenerować relacyjny model (przycisk *Engineer*). Szczególną uwagę zwrócić na tabelę *BD4_RACHUNEK*, w której to zostały dołożone dwie kolumny stanowiące klucze obce. Jeśli kolumny te mają nazwy *BD4_PRODUKT_nr_produktu* i *BD4_KLIENT_nr_klienta* należy wykonać czynności opisane w przypisie⁵ i ponownie wygenerować model relacyjny. W panelu Brower znaleźć w folderze *Relational Models* wygenerowany *Relational_1* i poprzez PPM na tej nazwie i *Properties* zmienić nazwę modelu relacyjnego na *rachunek* oraz *RDBMS Type* na *Oracle Database 12cR2*. Poprzez *Save* zapamiętać projekt.
5. *Data Modeler* umożliwia tworzenie obiektów fizycznego modelu danych (na poziomie konkretnego serwera bazodanowego). Poprzez panel Brower należy rozwinąć model relacyjny *rachunek* i na poziomie *Physical Models* przy pomocy PPM wybrać konkretny serwer (np. Oracle Database 12cR2). Odszukać w tej strukturze znane obiekty bazodanowe.
6. Utworzyć sekwencję *seq_klient*, która będzie zaczynała się od 100 i dawała przyrost kolejnych wartości o 10 oraz sekwencję *seq_rachunek* generującą wartości od 1 z inkrementacją 1. W obu przypadkach ustawić *Disable Cache* na Yes.
7. Rozwijając folder *Tables* oraz tabelę *BD4_RACHUNEK* w *Physical Models* uruchomić przy pomocy PPM na folderze *Triggers* tworzenie wyzwalacza (*trigger*). Wyzwalacz *tr_INS_rachunek* powinien wygenerować kolejną wartość dla kolumny *nr_rachunku* przy pomocy sekwencji *seq_rachunek*.

*Należy wypełnić odpowiednio pola na zakładce General: nazwę, akcję (*INSERT*), moment uruchomienia (*BEFORE*) i rodzaj wyzwalacza (*FOR EACH ROW*) oraz na zakładce Trigger Body wpisać kod PL/SQL realizujący funkcję wyzwalacza czyli:*

```
begin
    :NEW.nr_rachunku := seq_rachunek.nextval;
end;
```

⁴ Inną opcją określającą zasady kasowania jest *SET NULL*. Taka definicja powoduje, że kasowanie wiersza nadzędnego powoduje ustawienie w wierszach podrzędnych na pozycji klucza obcego wartości *null*. Ma to sens tylko w przypadku ustawienia opcji *Target Optional* na *Yes*

⁵ Przed wygenerowaniem modelu relacyjnego można zmienić ustawienia nazewnictwa poprzez przejście przy pomocy prawego przycisku myszy do właściwości diagramów (w panelu Brower -> Design *rachunek* -> PPM -> Properties -> Settings -> Naming Standard -> Templates). Wartość w polu Foreign Key zmienić na *{parent}_FK*, a w Column Foreign Key na *{ref column}*.

Prawidłowość kodu nie jest kontrolowana przez Data Modeler czyli ewentualne błędy mogą pojawić się w fazie zakładania obiektów w schemacie bazy danych.

8. Innym sposobem definiowania takich obiektów jak wyzwalacze, procedury i funkcje jest utworzenie tylko specyfikacji, pozostawiając kodowanie na późniejszy termin.
 Odnaleźć folder *Stored Procedures* w *Physical Models* i zdefiniować prototyp procedury o nazwie *pr_insert_produkty*, przy pomocy której wprowadzany będzie nowy produkt do tabeli *BD4_PRODUKT*. Po wpisaniu nazwy procedury w polu *Name* i zatwierdzeniu formularza pojawi się panel *SQL Developer* wraz z niekompletnym prototypem procedury. Należy go doprowadzić do poniższej postaci:

```
CREATE OR REPLACE PROCEDURE pr_insert_produkty
    (v_nr_produktu number,
     v_nazwa_produktu varchar2,
     v_cena_produktu number,
     v_rokProdukcji number default null,
     v_ranking number default 5) AS
BEGIN
    NULL;
    /*
        Procedura wstawia nowy produkt do tabeli BD4_PRODUKT.
        Należy sprawdzić, czy podany numer produktu już istnieje.
        Rok produkcji i ranking nie muszą być wypełnione.
    */
END;
```

i zamknąć ten panel "z krzyżyka" potwierdzając zmiany.

9. Przejść do ustawień *Tools / Preferences... / Data Modeler / DDL* i wyłączyć opcję *Include Default Settings in DDL* oraz *Include Logging in DDL*.
10. Przy pomocy funkcji *File / Export / DDL File / Generate* wygenerować dwa skrypty dla serwera Oracle. Pierwszy o nazwie *rachunek_create.ddl* (zakładka *Create Selection*) zawierający zdania SQL tworzące model bazy danych oraz drugi o nazwie *rachunek_drop.ddl* (zakładka *Drop Selection* - należy wybrać zaprojektowane obiekty: tabele, indeksy, klucze obce, sekwencje) zawierający dodatkowo zdania SQL usuwające tabele i powiązania między nimi. W obu przypadkach zawrzeć w skrypcie komentarze wprowadzone w modelu logicznym (*Include Comments*).

W trakcie generowania skryptów może pojawić się informacja o błędzie w projekcie. Odnaleźć w skrypcie szczegóły informacji i dokonać odpowiednich poprawek. Na przykład nazwa wygenerowanej relacji między tabelami może być za dłuża. Zmienić w modelu nazwy obu relacji, a następnie ponownie wygenerować skrypt lub ustawić opcje nazewnictwa na podstawie przypisu z poprzedniej strony.

W skrypcie tworzącym bazę danych zwrócić uwagę na różnice w definicjach kluczy obcych wynikające z przyjętej logiki modelu:

```
ALTER TABLE bd4_rachunek
ADD CONSTRAINT bd4_klient_fk FOREIGN KEY ( nr_klienta )
    REFERENCES bd4_klient ( nr_klienta );

ALTER TABLE bd4_rachunek
ADD CONSTRAINT bd4_produkty_fk FOREIGN KEY ( nr_produktu )
    REFERENCES bd4_produkty ( nr_produktu )
    ON DELETE CASCADE;
```

Relacja bd4_klient_fk jest restrykcyjna, co oznacza, że nie jest możliwe skasowanie klienta z ewidencji (obiekt nadzędny) jeśli istnieje wpis o zakupie przez niego co najmniej jednego produktu. Relacja bd4_produkt_fk umożliwia automatyczne skasowanie wszystkich transakcji dotyczących kasowanego produktu z katalogu (obiekt nadzędny).

Skrypt rachunek_drop.ddl zawiera w sobie zarówno zdania kasujące obiekty modelu jak i zdania tworzące te obiekty. Dla sprawności wdrożeniowej lepiej jest, aby te dwie funkcjonalności rozdzielić na poziomie skryptów. Dlatego też należy w skrypcie rachunek_drop.ddl pozostawić tylko zdania SQL usuwające obiekty bazodanowe ze schematu.

11. Przy pomocy narzędzia *SQL Developer* lub *SQLPlus* uruchamiać oba wygenerowane skrypty na serwerze oraz wypełnić utworzone tabele danymi wykorzystując dołączony do materiałów skrypt *rachunek_populate.sql*:

```
@c:\temp\rachunek_drop.ddl
@c:\temp\rachunek_create.ddl
@c:\temp\rachunek_populate.sql
```

12. Przetestować skuteczność działania więzów integralnościowych zdefiniowanych w modelu poprzez próby realizacji poniższych czynności i obserwując skutki tego działania:

- a. Podjąć próbę usunięcia z tabeli *BD4_KLIENT* Abackiego,
- b. Usunąć z tabeli *BD4_PRODUKT* obiektyw Obiektyw EF 24-105mm (przed i po tej operacji odczytać zawartość tabel *BD4_PRODUKT* i *BD4_RACHUNEK*),
- c. Usunąć z tabeli *BD4_PRODUKT* aparat Canon 6D Body (przed i po tej operacji odczytać zawartość tabel *BD4_PRODUKT* i *BD4_RACHUNEK*),
- d. Zaewidencjonować (przy założeniu istnienia wyzwalacza *tr_INS_rachunek*) nową transakcję dla nieistniejącego klienta oraz sprawdzić efekty używając sekwencji zdań:

```
select seq_rachunek.curval from dual;
select seq_klient.curval from dual;

insert into bd4_rachunek
    (data_sprzedazy,
     nr_klienta,
     nr_produktu,
     ilosc_produktu)
values
    (sysdate,
     105 ,
     3021,
     3);

select seq_rachunek.curval from dual;
select * from bd4_klient;
select * from bd4_rachunek;
```

- e. Wykonać zdanie SQL *rollback*.

13. Zmodyfikować model logiczny uwzględniający strategię kasowania polegającą na ustawianiu właściwości *Delete Rule* na *Set NULL* w przypadku chęci kasowania obiektu nadzędnego, na

przykład dla związku *produkt_rachunek*⁶, zaimplementować ten model i przetestować skuteczność tych modyfikacji.

14. Przejść do właściwości *Data Modeler* (*Tools/Preferences*) i zapoznać się i ewentualnie zmienić niektóre ustawienia środowiska, np.:

- ścieżki dostępu do katalogów roboczych (*Data Modeler*),
- standardową notację w modelu logicznym (*Data Modeler / Diagram / Logical Model*),
- kierunek strzałek na diagramie modelu relacyjnego (*Data Modeler / Diagram / Relational Model*) - zaleca się ustawienie opcji *From Primary Key to Foreign Key*,
- standardowy RDBMS (*Data Modeler / Model*) – Oracle Database 12cR2,
- ograniczenie wyboru typów danych przy projektowaniu encji (*Data Modeler / Model*):

Columns & Attributes Defaults

Nulls Allowed

Datatype: Domain Logical Distinct Structured Collection

Preferred Logical Types

- Date
- Datetime
- NUMERIC
- VARCHAR

Aby powyższe ustawienie stało się aktywne należy w trakcie projektowania encji ustawić opcję *Preferred*:

Attribute Properties

Name	Attribute_2
Data Type	<input type="radio"/> Domain <input checked="" type="radio"/> Logical <input type="radio"/> Distinct <input type="radio"/> Structured <input type="radio"/> Collection
Source Type	VARCHAR <input checked="" type="checkbox"/> Preferred

15. Przy pomocy przycisku *Export* (znajdującego się na formularzu *Preferences / Data Modeler*) wyeksportować do własnego folderu opracowane opcje. Mogą one być potem importowane w innej instalacji *Data Modeler* z tego samego miejsca przyciskiem *Import*.

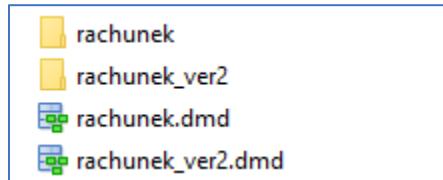
16. Zakończyć pracę z *Data Modeler*.

⁶ Należy mieć na uwadze fakt, że struktura encji musi umożliwiać realizację takiej strategii, to znaczy właściwość *Target Optional* musi być zaznaczona (co oznacza No Mandatory czyli dopuszcza się *null*).

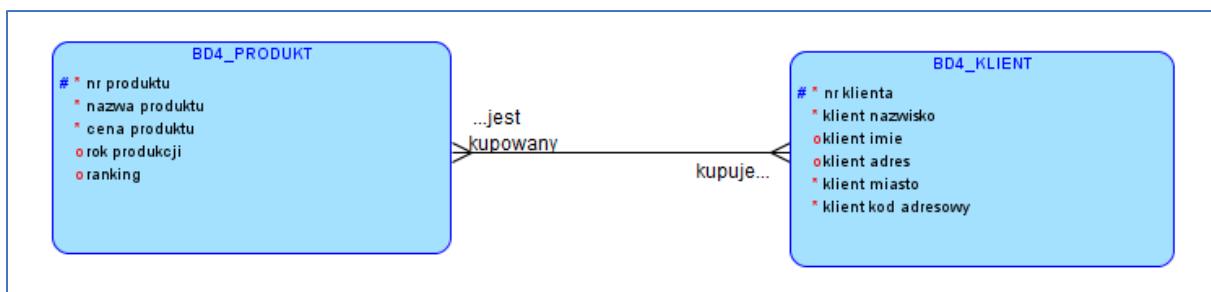
III. Projektowanie modelu bazodanowego z relacją M:N (many to many)

Należy opracować drugą wersję modelu *rachunek* (wcześniej zaimplementowanego) w oparciu o takie same założenia. Istotną różnicą będzie przedstawienie modelu logicznego w postaci dwóch encji (*Klient* i *Produkt*) będących ze sobą w związku M:N.

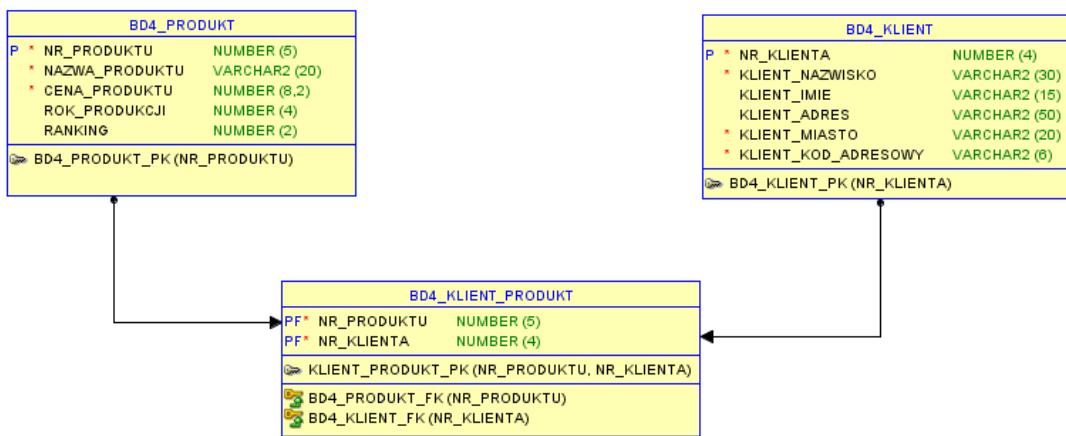
1. W *Data Modeler* otworzyć opracowany logiczny model *rachunek*, a następnie zapamiętać go (save as...) pod nazwą *rachunek_ver2*.



2. W tej wersji modelu logicznego usunąć encję *BD4_RACHUNEK*. Automatycznie zostaną usunięte oba zdefiniowane uprzednio związki między encjami. Usunąć model relacyjny poprzez *Browser / Relational Model* oraz założyć nowy i nadać mu powtórnie nazwę *rachunek*.
3. Między encjami *BD4_PRODUKT* i *BD4_KLIENT* utworzyć związek wiele do wielu (M:N) i nazwać go *klient_produkt*. Określić *Name of Source* na "...jest kupowany", a *Name of Target* na "kupuje...". *Source Optional* i *Target Optional* powinny być ustawione na *No* (niezaznaczone), podobnie jak opcja *Use surrogate keys*.



4. Wygenerować model relacyjny. Powstała trzecia tabela, w której klucz główny jest kluczem złożonym zbudowanym na kolumnach będących kluczami głównymi w pozostałych tabelach:



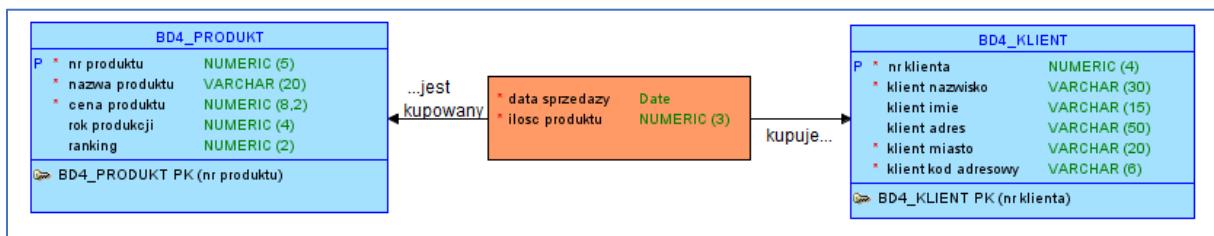
5. Powrócić do edycji modelu logicznego. Otworzyć właściwości związku *klient_produkt* (np. Browser / Logical Model / Relations) i zdefiniować dodatkowe atrybuty tego związku (przejść do Attributes):

- *data sprzedazy* typu Date obowiązkowy,
- *ilosc produktu* typu Numeric(3) obowiązkowy

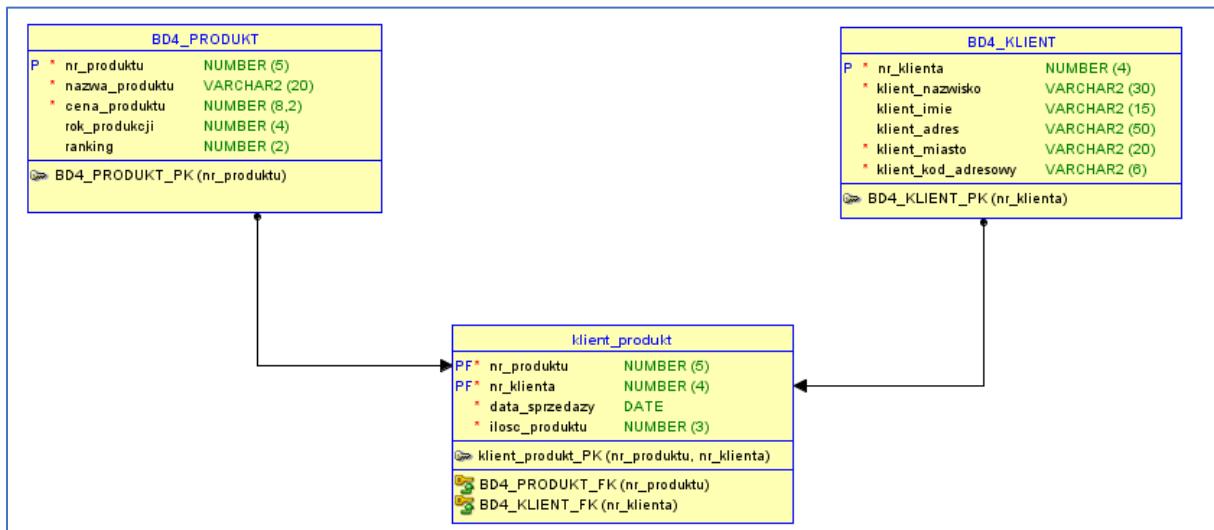
, zatwierdzić te definicje, a następnie w dowolnym miejscu panelu przy pomocy PPM wybrać Show / Relationship Attributes.



Zmienić notację diagramu z notacji Barkera na Bachmana (w panelu PPM i Notation).

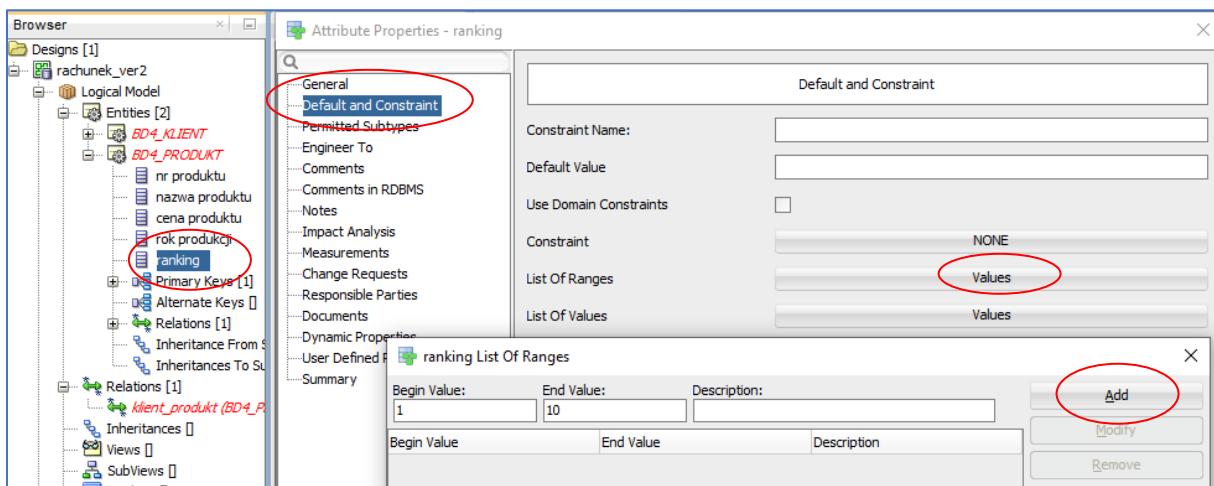


6. Powtórnie wygenerować model relacyjny:

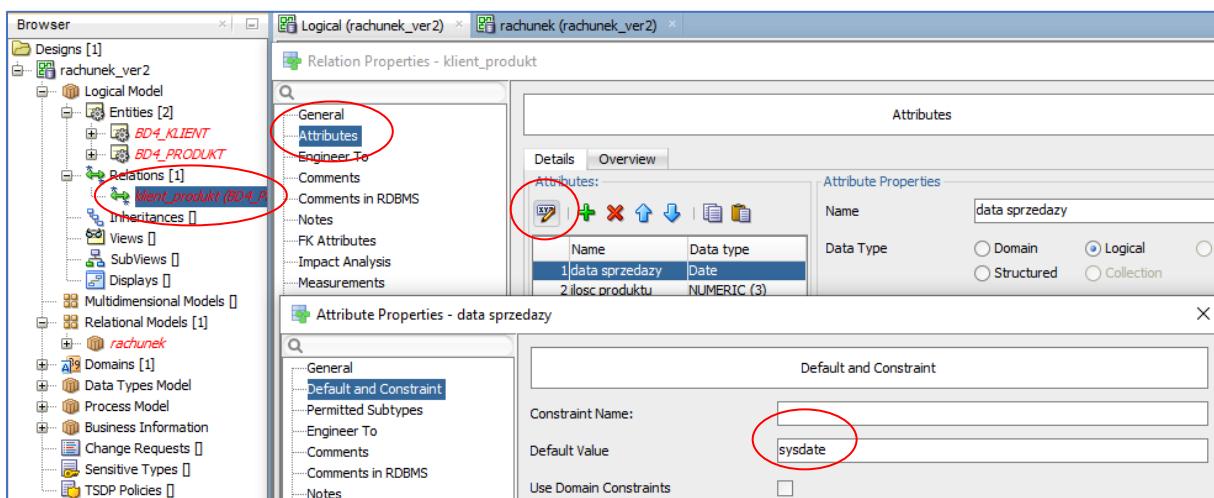


Dodatkowe definicje związku między encjami zostały uwzględnione w tabeli pośredniczącej (intersekcji).

7. Powrócić do edycji modelu logicznego. Wybrać właściwości atrybutu *ranking* w encji *BD4_PRODUKT* i ustawić zakres dopuszczalnych wartości na przedział [1..10]:



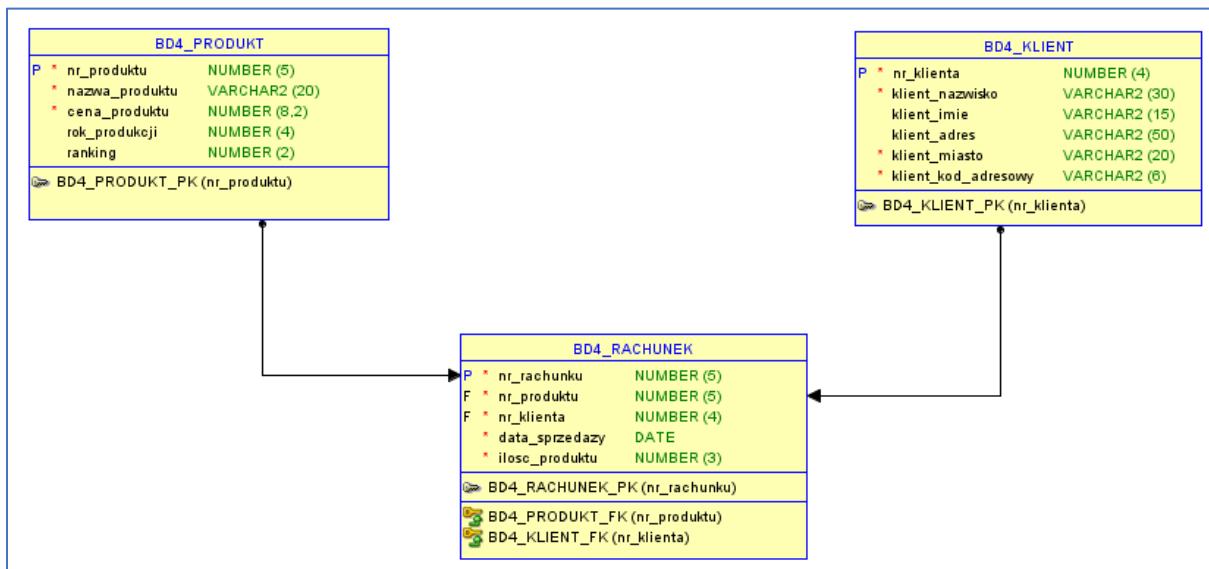
8. Podobnie dla atrybutu *data sprzedaży* ustawić bieżącą datę jako domyślną:



I wygenerować model relacyjny.

9. Utworzony model relacyjny zawiera tabelę pośrednią (*klient_produkty*), której kluczem głównym jest para numer klienta i numer produktu. Jest to wada tego modelu, gdyż oznacza to, że dany klient może kupić dany produkt tylko jeden raz, a to nie jest zgodne z rzeczywistością. Należy ten model zmodyfikować wprowadzając do tabeli *klient_produkty* nowy klucz główny (*nr_rachunku* NUMERIC(5)) umieszczając go dodatkowo na pierwszej pozycji w tabeli oraz usuwając tę właściwość z kolumn *nr_klienta* i *nr_produktu* (te modyfikacje można wykonać otwierając właściwości tabeli).
10. Na koniec zmienić nazwę tabeli na *BD4_RACHUNEK* (na formularzu *General*) oraz nazwę definicji klucza głównego (na formularzu *Primary Key*) z *klient_produkty_PK* na *BD4_RACHUNEK_PK*.

Ostateczna postać modelu relacyjnego powinna wyglądać tak:



Należy zaznaczyć, że po tych modyfikacjach nie jest gwarantowane ponowne prawidłowe generowanie modelu relacyjnego na podstawie modelu logicznego. Innymi słowy, jeśli dokonalibyśmy modyfikacji w modelu logicznym i ponownie wygenerowali model relacyjny to należałoby ponownie dokonywać zmian w wygenerowanym modelu relacyjnym, które zostały zrobione w punkcie 9 i 10.

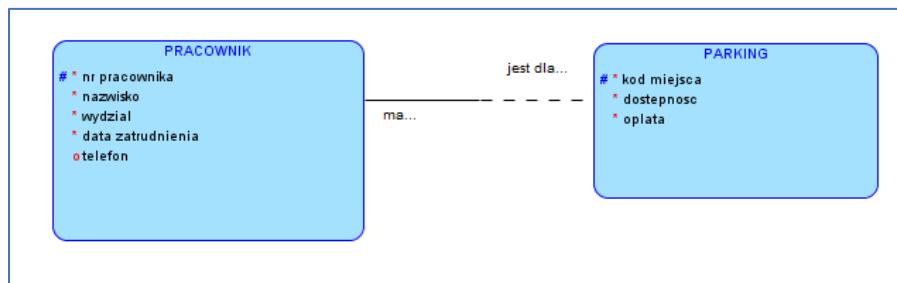
11. Wygenerować nową wersję skryptu DDL (*rachunek_create_ver2.ddl*) i ocenić jego zgodność z czynnościami wykonanymi na projektowanym modelu.
12. Wprowadzić przykładowe dane do tabel (zmodyfikowanym odpowiednio skryptem *rachunek_populate_ver2.sql* na podstawie *rachunek_populate.sql*)⁷. Sprawdzić skuteczność działania opracowanych restrykcyjnych strategii kasowania wierszy w tabelach.

IV. Projektowanie modelu bazodanowego z relacją 1:1 (one to one)

Związek 1:1 definiuje zasadę, że w modelu logicznym pojedynczy egzemplarz jednej encji jest w związku z co najwyżej jednym egzemplarzem drugiej encji. Jak łatwo zauważyc, trudno w takim przypadku mówić o hierarchii obiektów (nadzędny – podrzędny).

Załóżmy, że firma dla swoich pracowników organizuje na swoim terenie miejsca parkingowe. Zakłada się, że każdy pracownik obligatoryjnie ma przydzielone jedno i tylko jedno takie miejsce.

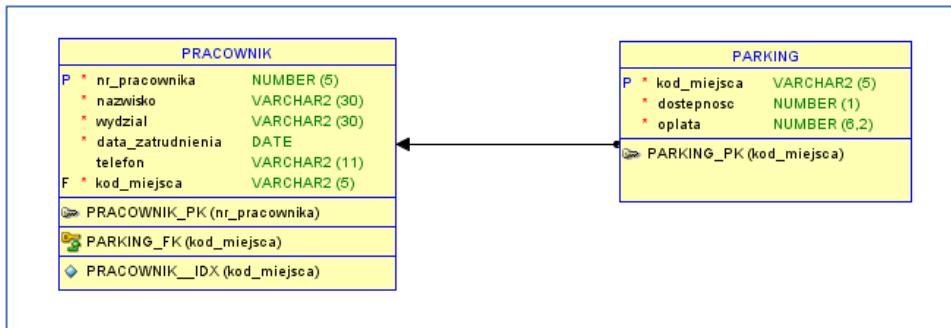
Model logiczny mógłby wyglądać tak (budowa związku 1:1 od *PARKING* do *PRACOWNIK*)⁸:



⁷ Brak jest takich obiektów jak sekwencje i wyzwalacz oraz zdefiniowana jest domyślna wartość data sprzedaży.

⁸ Należy usunąć możliwość użycia klucza sztucznego (opcja *Use surrogate keys* ma być niewybrana).

, a odpowiadający mu model relacyjny:



W tak wygenerowanym modelu relacyjnym należy zwrócić uwagę na trzy aspekty:

1. W tabeli *PRACOWNIK* został umieszczony klucz obcy pochodzący z tabeli *PARKING*. Właściwość istnienia jest ustawiona na *obowiązkowość* (*), co przekłada się na definicję kolumny:

.....
kod_miejsca VARCHAR2(5) **NOT NULL**
.....

A to oznacza, że każdy pracownik, z urzędu, ma przydzielone miejsce parkingowe.

2. W tabeli *PRACOWNIK* automatycznie został wygenerowany indeks *PRACOWNIK_IDX* oparty na kolumnie klucza obcego *kod_miejsca*:

*CREATE UNIQUE INDEX pracownik_idx ON
PRACOWNIK (kod_miejsca ASC);*

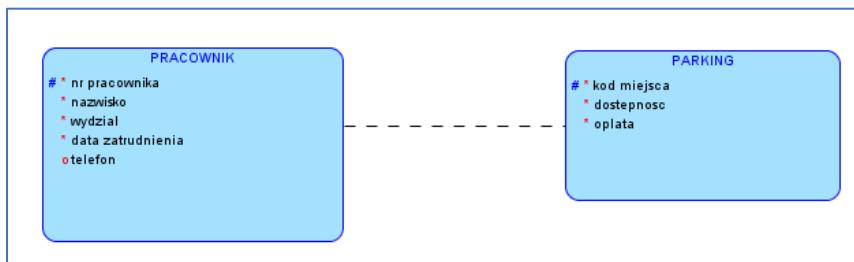
Jest to indeks unikalny, co oznacza, że w kolumnie *kod_miejsca* tabeli *PRACOWNIK* nie może wystąpić więcej niż raz ta sama wartość. Spełnia to założenie przydziału każdemu pracownikowi jednego miejsca parkingowego.

3. Gdyby zmienić założenie na opcjonalność przydziału miejsca parkingowego czyli pracownik nie musi, ale może posiadać takowe, wystarczy w modelu logicznym ustawić w definicji związku *Target Optional* na Yes oraz jako *Dominant Role* wybrać *PARKING*. Wtedy w wygenerowanym na nowo modelu relacyjnym kolumna *kod_miejsca* otrzyma atrybut *null*:

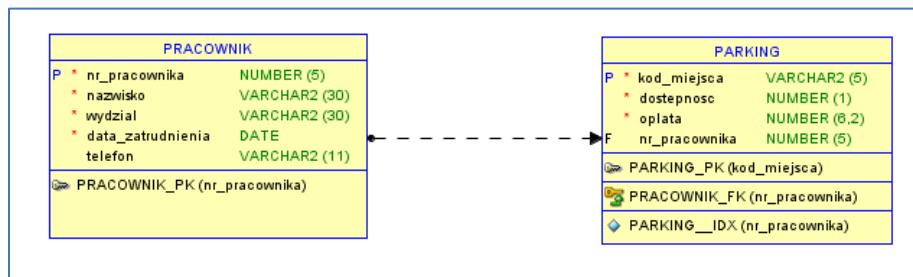
.....
kod_miejsca VARCHAR2(5)
.....

Dopuszczenie istnienia wartości *null* w kolumnie *kod_miejsca* nie ma wpływu na działanie unikalnego indeksu *pracownik_idx*, gdyż wystąpienia *null*, w takim przypadku, nie podlegają analizie na powtarzalność wartości.

Warto rozważyć sytuację odwrotną do omawianej do tej pory budując związek od *PRACOWNIK* do *PARKING*:



, a wygenerowany model relacyjny tak:



Nadal spełnione są założenia, że pracownik może (ale nie musi) mieć przydzielone co najwyżej jedno miejsce parkingowe (istnieje unikalny indeks *PARKING_IDX*).

Istotna różnica może się pojawić w przypadku chęci skasowania pracownika z ewidencji. W pierwszym wariantie modelu można uznać nadrzędność tabeli *PARKING* nad tabelą *PRACOWNIK*, gdyż klucz obcy jest usadowiony w tej drugiej. Skasowanie wiersza w tabeli podrzędnej (*PRACOWNIK*) nie zakłóca spójności bazy danych i może być bez przeszkód wykonane.

W wariantie drugim mamy sytuację odwrotną. Tabela *PRACOWNIK* jest tabelą nadrzędową, gdyż klucz obcy znajduje się w tabeli *PARKING*. Kasowanie wiersza nadzawanego nie jest takie oczywiste. Jeśli trzeba skasować pracownika, który ma przydzielone miejsce parkingowe to system zarządzania bazą danych do tego nie dopuści z racji zachowania spójności bazy danych, jeśli relacja między tabelami będzie zdefiniowana jako restrykcyjna (RESTRICT). Nielogicznym biznesowo byłoby najpierw skasowanie odpowiedniego wiersza w tabeli *PARKING*, a następnie pozostającego z nim w relacji pracownika w tabeli *PRACOWNIK* z racji utraty danych o miejscu parkingowym (samym w sobie).

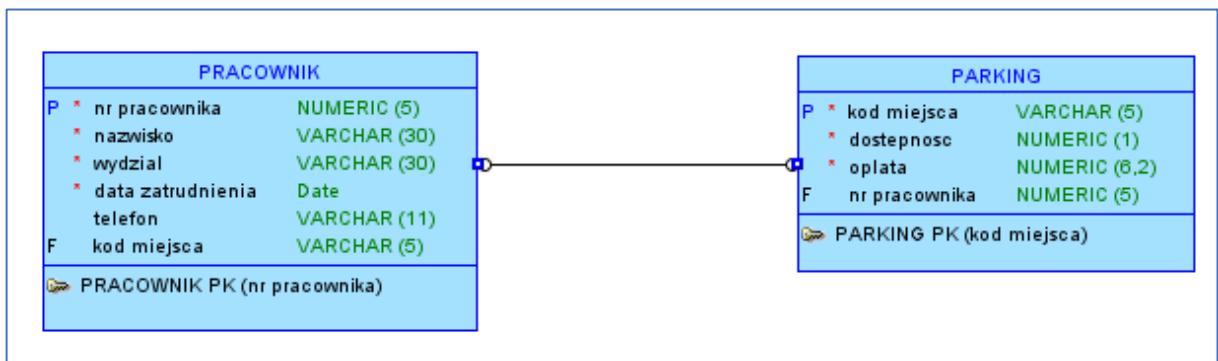
Najlepszym, w takiej sytuacji, rozwiązaniem jest modyfikacja relacji między tabelami poprzez zmianę zasad dotyczącej kasowania wiersza nadzawanego *Delete Rule* z RESTRICT na Set NULL. Można to zmienić w modelu relacyjnym w definicji klucza obcego *pracownik_fk* zmieniając strategię *Delete Rule* na SET NULL. W wygenerowanym skrypcie implementującym model objawi się to zdaniem:

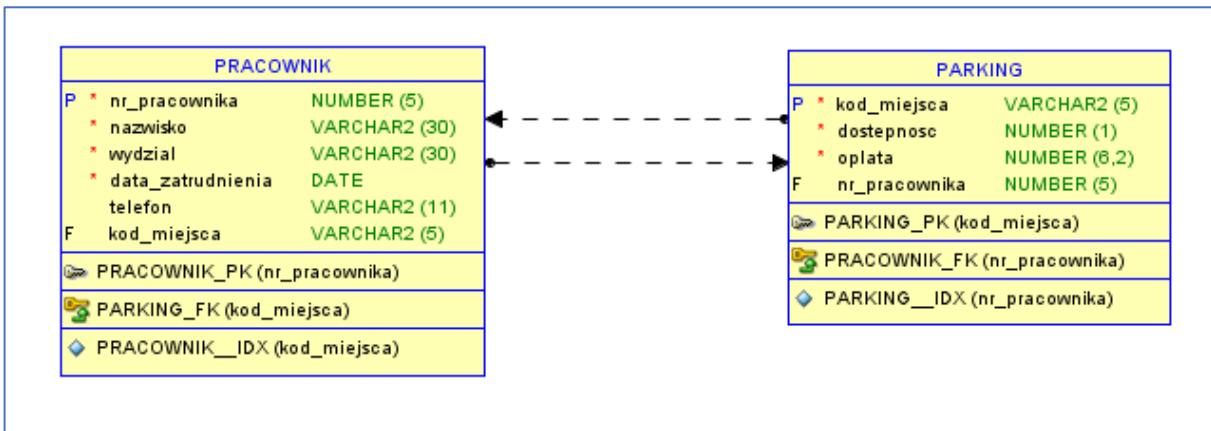
```

ALTER TABLE parking
    ADD CONSTRAINT pracownik_fk FOREIGN KEY ( nr_pracownika )
    REFERENCES pracownik ( nr_pracownika )
    ON DELETE SET NULL;
  
```

Teraz, kasując pracownika w tabeli *PRACOWNIK* równocześnie w odpowiadającym mu wierszu tabeli *PARKING* w kolumnie *nr_pracownika* zostanie wstawiony *null*, ale wiersz pozostanie.

Inną metodyką modelowania związków *one_to_one* jest wprowadzanie równoczesne do obu tabel kluczy obcych naprzemiennie. Należy w modelu logicznym w definicji związku między encjami zlikwidować dominującą rolę jednej z nich (*Dominant Role* ustawić na None) oraz *Source* i *Target Optional* na Yes.





Każdy pracownik może, ale nie musi, mieć przydzielone co najwyżej jedno miejsce parkingowe, a każde miejsce parkingowe może, ale nie musi, być przydzielone co najwyżej jednemu pracownikowi.

W takim modelu problemem jest zagwarantowanie dwustronności relacji między wierszami obu tabel. Jeśli w tabeli PRACOWNIK Abacki ma przydzielone miejsce parkingowe A-101, to jak zapewnić, aby w tabeli PARKING miejsce parkingowe A-101 było przydzielone Abackiemu? Potrzebne są rozwiązania programowe (wyzwalacze, procedury), których kod może zapewnić poprawność danych.

Uwagi końcowe:

1. W modelach relacyjnych najczęściej występują związki *one_to_many* oraz rzadziej *one_to_one*. Natomiast nie mogą występować związki *many_to_many*, gdyż takiego związku nie można implementować. Przetwarza się je na dwa związki *one_to_many*.
2. Związek *one_to_one* można zrekonfigurować w ten sposób, że wszystkie atrybuty obu encji umieścić w jednej:

PRACOWNIK	
P	* nr_pracownika NUMERIC(5)
*	nazwisko VARCHAR(30)
*	wydział VARCHAR(30)
*	data_zatrudnienia Date
telefon	VARCHAR(11)
kod_miejsca	VARCHAR(5)
oplata NUMERIC(6,2)	
PRACOWNIK_PK (nr_pracownika)	

Wybór odpowiedniej techniki zależny jest od konkretnego obszaru modelowania i nie sprawdza się w każdym przypadku. Tabela powstała na podstawie powyższej encji nie jest dobrym przykładem, gdyż nie jest tabelą znormalizowaną (*oplata* nie jest zależna funkcjonalnie od *nr_pracownika* tylko od *kod_miejsca*).

3. Obszarami zastosowania związków *one_to_one* mogą być dane częściowo wrażliwe. Tabela danych pracowników zawierająca dane typowe, jak również dane wrażliwe może być zdekomponowana na dwie tabele. Tabela z danymi wrażliwymi podlegająca specjalnej ochronie może być szyfrowana.
4. Innym przykładem może być sytuacja, gdy tabela zawiera dane bardzo często przetwarzane i rzadko przetwarzane, ale pozostające ze sobą w związku *one_to_one*. Z analizy wydajności działania bazy można wysnuć wniosek, że opłaca się podzielić tę tabelę na dwie oddzielne, aby zmniejszyć wielkość transferów z/do pamięci dyskowej, a przez to poprawić wydajność bazy.

V. Informacje dodatkowe i uzupełniające

Do tej pory omówione zostały podstawowe zasady tworzenia modelu logicznego i relacyjnego przy użyciu związków i relacji typu *one_to_many* i *many_to_many* w najczęściej spotykanych zastosowaniach.

Poniżej zostaną przedstawione inne zasady modelowania wynikające z ogólnej teorii modelowania baz danych oraz z możliwości Oracle Data Modeler.

Jak zaznaczono wcześniej model logiczny składa się z encji i związków zachodzących między encjami. Każdy związek posiada trzy cechy: stopień związku, typ asocjacji (kardynalność) oraz cechę istnienia (klasę przynależności).

Stopień związku określa liczbę encji połączonych związkiem. Wyróżnia się związki unarne (łączące encję samą ze sobą), binarne (łączące dwie encje) oraz n-arne (łączące n encji).

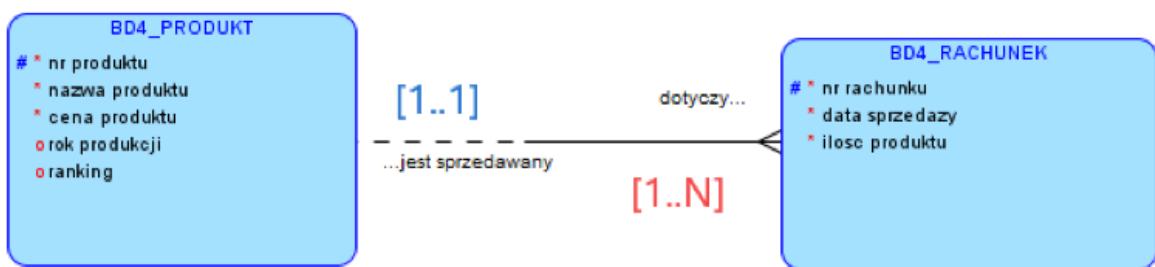
Typ asocjacji określa ile egzemplarzy jednej encji może wchodzić w związek z jednym egzemplarzem drugiej encji. Wyróżnia się typy: *one_to_one*, *one_to_many* i *many_to_many*.

Cecha istnienia określa czy dany związek jest opcjonalny czy obowiązkowy.

W tym materiale omówione zostaną tylko związki binarne wszystkich typów asocjacji i obu cech istnienia.

Związek one to many

Związek 1:N definiuje zasadę, że w modelu logicznym pojedynczy egzemplarz encji nadzędnej jest w związku z pewną liczbą egzemplarzy encji podrzędnej. Czyli jeden produkt **może** występować na wielu rachunkach, ale każdy rachunek **musi** dotyczyć jednego produktu. W wielu notacjach modelowania fakt ten zobrazowuje się na diagramach przy pomocy pary [1..N] lub podobnej.



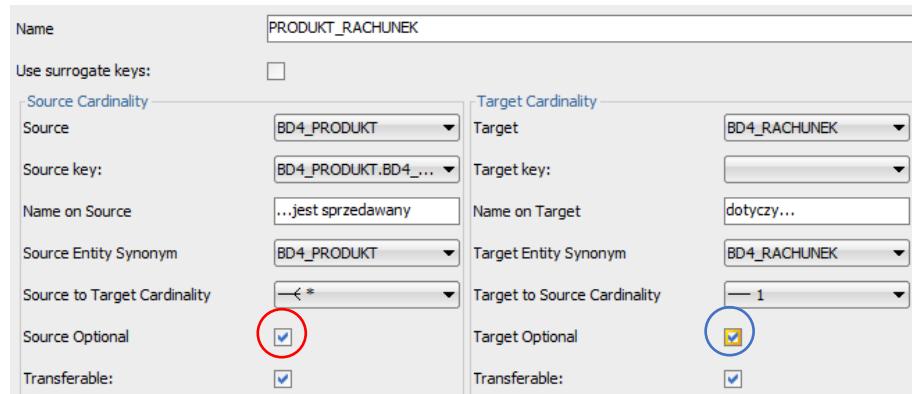
Oznaczenie **[1..N]** określa, że każdy produkt musi mieć co najmniej jedno wystąpienie w encji *rachunek*, a oznaczenie **[1..1]** określa, że z każdym egzemplarzem encji *RACHUNEK* związany jest jeden i tylko jeden produkt. Innymi słowy – nie może egzystować w ewidencji produktów produkt, który nie został kupiony, z racji definicji **[1..N]** oraz każdy rachunek musi zawierać jeden produkt (**[1..1]**).

W omawianym przykładzie jest to sprzeczne z logiką biznesową, gdyż katalog produktów może zawierać produkty nie kupione do tej pory. Aby być w zgodzie z taką logiką należy zastosować kolejną właściwość związków między encjami, a mianowicie istnienie (*opcjonalność / obowiązkowość*) związku.

Obowiązkowość wyrażana jest zapisem **[1..N]**, co oznacza, że każdy egzemplarz encji nadzędnej musi być w związku z co najmniej jednym egzemplarzem encji podrzędnej. Natomiast opcjonalność oznaczana jest jako **[0..N]** czyli egzemplarz encji nadzędnej nie musi być w związku z żadnym egzemplarzem encji podrzędnej (mogą istnieć w ewidencji produktów te, które nie były jeszcze kupione ani razu). I taka definicja jest odpowiednia dla omawianego przykładu.

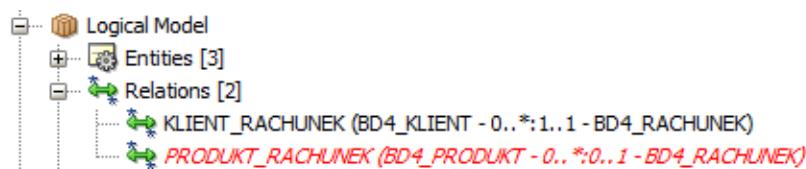
Podobnie oznaczenie [1..1] określa, że każdy egzemplarz encji podrzędnej musi być związany z dokładnie jednym egzemplarzem encji nadrzędnej, a oznaczenie [0..1] – że ten związek może zachodzić, ale nie musi. Innymi słowy – dopuszcza się istnienie transakcji bez określonego produktu.

W środowisku Oracle Data Modeler tę właściwość definiuje się przy pomocy ustawienia *Source Optional* i *Target Optional*:



Zmieniając wartości obu ustawień można uzyskać cztery możliwe warianty istnienia związku między dwiema encjami. Zmiany te można obserwować rozwijając w *Browser Logical Model / Relations*.

Dla powyższych ustawień związek *PRODUKT_RACHUNEK* jest zdefiniowany następująco:



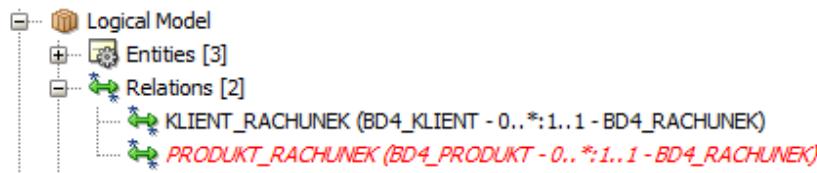
, co odpowiada notacji: [0..N] i [0..1].

Po wygenerowaniu modelu relacyjnego oraz skryptu ddl, jego fragment zawiera poniższą definicję:

```
CREATE TABLE bd4_rachunek (
    nr_rachunku      NUMBER(4) NOT NULL,
    data_sprzedazy   DATE NOT NULL,
    ilosc_produktu   NUMBER(3) NOT NULL,
    nr_produktu      NUMBER(5),
    nr_klienta        NUMBER(4) NOT NULL
);
```

, czyli dopuszcza się nieokreśloność produktu w transakcji zakupu.

Kombinacja [0..N] i [1..1] czyli *Source Optional*: wybrana, a *Target Optional*: nie wybrana daje inny efekt:



, zmiana jest widoczna na diagramie modelu relacyjnego oraz w wygenerowanym skrypcie:

```
CREATE TABLE bd4_rachunek (
    nr_rachunku      NUMBER(4) NOT NULL,
    data_sprzedazy   DATE NOT NULL,
    ilosc_produktu   NUMBER(3) NOT NULL,
    nr_produktu      NUMBER(5) NOT NULL,
    nr_klienta        NUMBER(4) NOT NULL
);
```

, czyli każda transakcja zakupu musi dotyczyć określonego produktu.

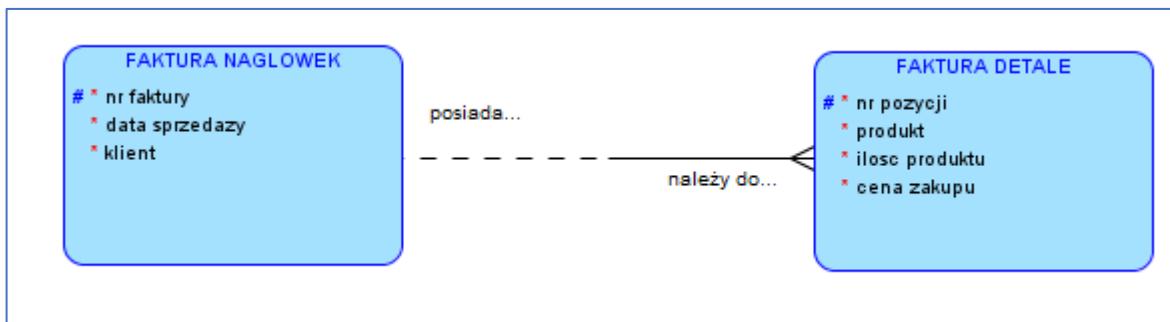
W podobny sposób można analizować dwie pozostałe kombinacje ustawień istnienia związku czyli [1..N] i [0..1] oraz [1..N] oraz [1..1].

Narzędzia służące modelowaniu baz danych nie mają środków umożliwiających zdefiniowanie związku: „każdy element encji nadzędnej musi być w związku z co najmniej jednym egzemplarzem encji podrzędnej” ([1..N]), czyli w katalogu produktów są tylko takie produkty, które chociaż raz były kupione. Tak postawiony problem rozwiązuje się metodami programowymi na poziomie serwera bazodanowego (wyzwalacze, procedury) lub na poziomie aplikacji.

Słaba encja (weak entity)

Słaba encja to szczególny typ encji, w której identyfikator (atrybut kluczowy) składa się (przynajmniej częściowo) z atrybutów kluczowych innych encji.

Przykładem niech będzie fragment modelu logicznego fakturowania wielopozycyjnego:

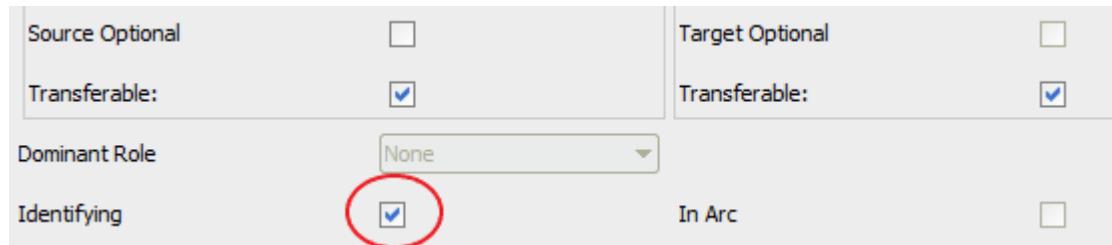


Faktura składa się z nagłówka oraz pozycji faktury. Jeden egzemplarz encji *FAKTURA_NAGLOWEK* jest w związku 1:N z wieloma egzemplarzami encji *FAKTURA_DETALE*.

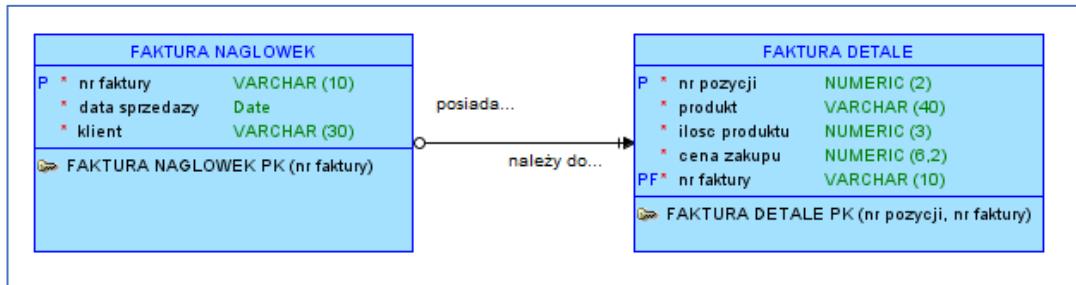
Obie encje mają swoje własne identyfikatory. #nr faktury jednoznacznie identyfikuje każdy egzemplarz encji *FAKTURA_NAGLOWEK*, ale #nr pozycji w odniesieniu do encji *FAKTURA_DETALE* – nie, gdyż dla każdej faktury numeracja pozycji zaczyna się od wartości 1 (np. nr faktury 100/AB/19 ma pozycje 1,2 i 3, a nr 101/AB/19 – 1,2,3 i 4). Dopiero para tych identyfikatorów w sposób jednoznaczny określa egzemplarz encji *FAKTURA_DETALE*.

Aby to osiągnąć w Oracle Data Modeler należy, określając związek między encjami, wybrać odpowiednią definicję związku (*1:N Relation Identifying*).

We właściwościach zostanie to uwzględnione poprzez zaznaczenie opcji *Identifying*:



Zmieniając wygląd diagramu tego modelu logicznego z notacji *Barkera* na *Bachmana*:

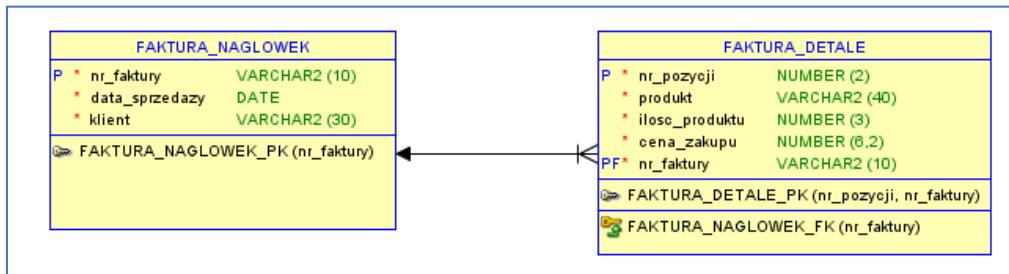


można zauważyć, że w prawidłowy sposób została zaprojektowana encja detali faktury. Identyfikator jest złożony i częściowo zależny od identyfikatora encji nagłówka faktury.

I to jest istotą encji słabych.

Chcąc wygenerować prawidłowo model relacyjny należy w obu encjach poprzez *Entity Properties / General* usunąć możliwość tworzenia klucza sztucznego (*Create Surrogate Key*), jeśli z jakiś względów ta możliwość jest zaznaczona.

Wygenerowany model może wyglądać tak:



Dla porządku można zarówno w modelu logicznym (notacja *Bachmana*) i w modelu relacyjnym przenieść *nr_faktury* w detalicach faktury na pierwszą pozycję (zarówno w strukturze tabeli jak i w definicji klucza głównego).



VI. Projektowanie modelu uwzględniającego dziedziczenie obiektów i typy strukturalne

Opracować projekt logiczny OSOBA w oparciu o poniższe założenia:

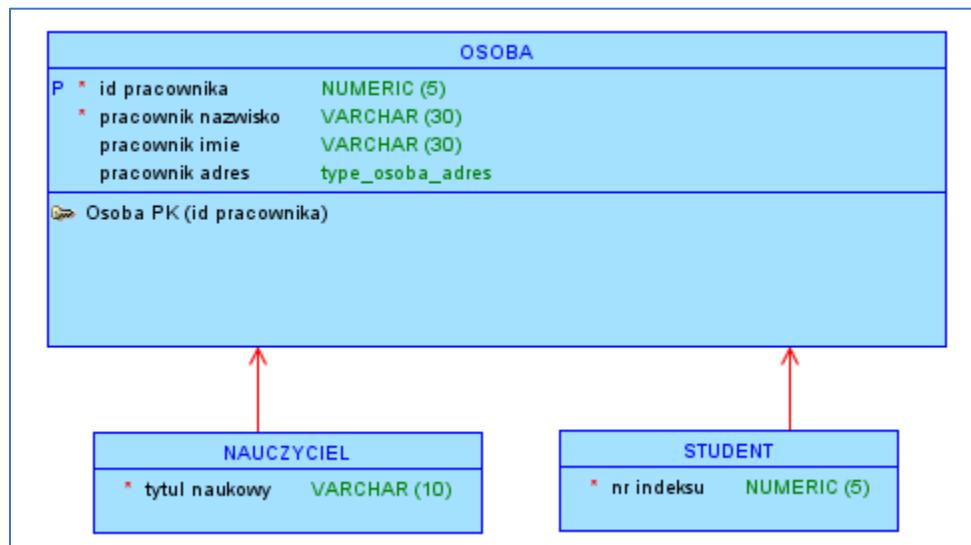
Model logiczny powinien zawierać trzy encje: OSOBA, NAUCZYCIEL i STUDENT, przy czym encja OSOBA ma pełnić rolę obiektu nadzawanego, co oznacza, że dwie pozostałe encje mają, oprócz posiadania własnych atrybutów, dziedziczyć jej atrybuty. Dodatkowo należy zdefiniować własny typ strukturalny określający adres zamieszkania. Każda z osób (zarówno nauczyciele jak i studenci) może posiadać kilka kont w funkcjonujących systemach informatycznych.

1. W celu utworzenia własnego typu strukturalnego należy w lewym panelu odnaleźć folder *Data Types Model* i przy pomocy prawego przycisku myszy wybrać *Show*, a następnie z głównego menu odpowiednią ikonę na pasku narzędziowym (*New Structured Type*) i kliknąć w panelu roboczym.
2. Nadać nazwę tworzonemu obiekowi: *type_osoba_adres* i przejść do definiowania atrybutów według poniższego rysunku analogicznie do definiowania atrybutów encji:

type_osoba_adres	
ulica	VARCHAR (25)
M miasto	VARCHAR (15)
kod	VARCHAR (10)

Litera **M** (Mandatory) przy atrybutie oznacza, że musi być on zawsze określony.

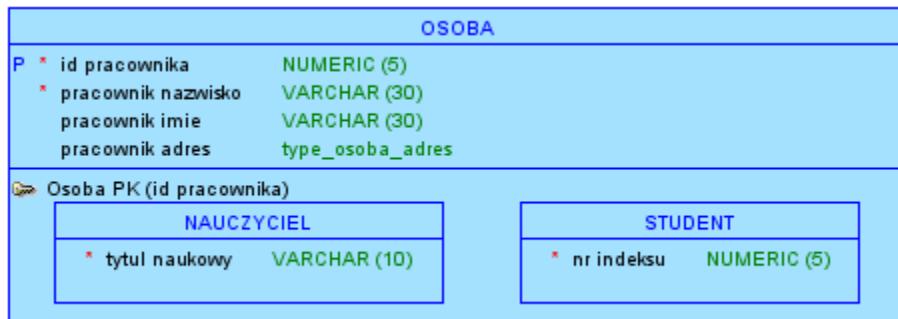
3. Zdefiniować w standardowy sposób trzy encje OSOBA, NAUCZYCIEL i STUDENT według poniższych diagramów:



Dla atrybutu *pracownik adres* wybrać dla *Data Type* opcję *Structured*, a następnie rozwiniąć *Source Type* i wybrać zdefiniowany typ.

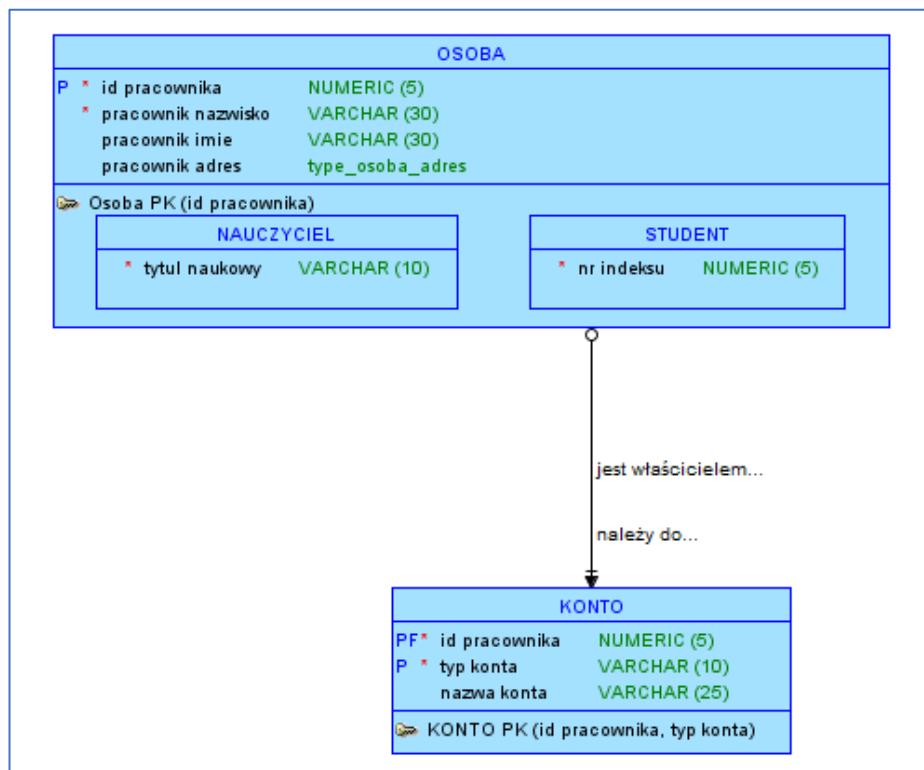
4. Przystąpić do modyfikacji encji NAUCZYCIEL i STUDENT polegającej na określeniu obiektu, z którego tworzone encje będą dziedziczyć atrybuty. Realizuje się to poprzez wybór obiektu nadzawanego w polu *Super Type* we właściwościach modyfikowanej encji (*Properties / General*).

5. Sposób prezentacji dziedziczenia obiektów można zmienić. W tym celu należy w panelu roboczym kliknąć prawym przyciskiem myszy poza zdefiniowanymi obiektami i wybrać funkcję *Notation / Box-In-Box Presentation*, a następnie poprzez zabiegi modyfikujące wygląd doprowadzić do ostatecznej postaci:



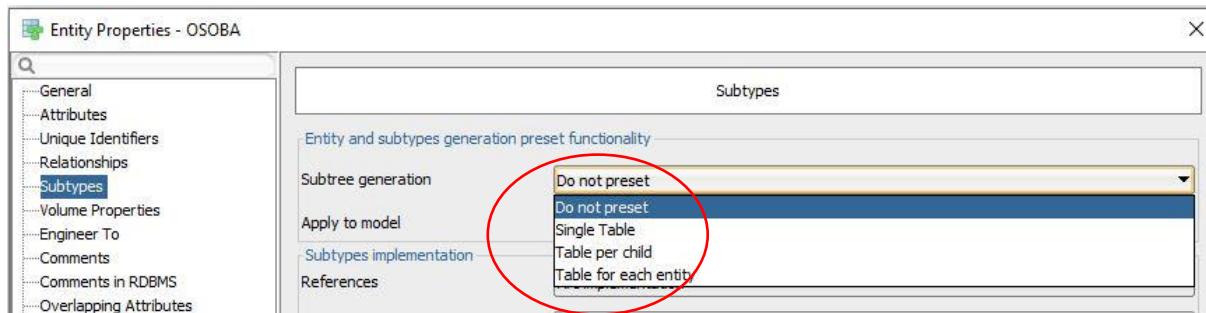
Sprawdzić sposób prezentacji w obu typach notacji (Bachman i Barker).

6. Zamodelować encję KONTO odzwierciedlającą przydzielanie konta komputerowego pracownikom uczelni (nauczycielom i studentom), przy czym dopuszcza się posiadanie kilku kont przez jedną osobę⁹. Encja posiada dwa atrybuty: *typ_konta varchar(10)* jako unikalny identyfikator i nieobowiązkowy *nazwa_konta varchar(25)*. Dodatkowo należy zdefiniować dopuszczalne wartości atrybutu *typ_konta*, np.: *mail, oracle, teams* i *extranet*. Można to zrealizować we właściwościach atrybutu (*Attribute Properties*) i w zakładce *Default and Constraints* nazywając te więzy np. *typ_konta_ch* oraz edytując pozycję *List Of Values* wprowadzając założone nazwy typów kont.



⁹ W definicji encji KONTO nie należy definiować atrybutu *id_pracownika*. Po określaniu związku (1:N) między encjami identyfikator pracownika stanie się kluczem obcym. Należy wtedy dodatkowo zdefiniować go jako część klucza głównego i umieścić na odpowiedniej pozycji. Można to zrobić ustawiając w definicji relacji *osoba_konto* opcję *Identifying* oraz kasując opcję *Use surrogate keys*.

7. Przed przystąpieniem do generowania relacyjnych modeli należy zwrócić uwagę na ustawienie opcji odpowiedzialnej za sposób implementacji tabel OSOBA, NAUCZYCIEL i STUDENT. W tym celu dla encji OSOBA z głównych właściwości encji odczytać wartości w polu *Subtree generation* (*General / Subtypes*).



Istnieje kilka sposobów transformacji takiego modelu logicznego.

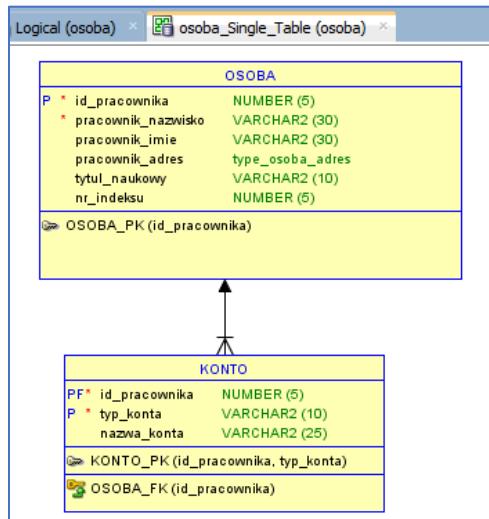
Poniżej zostanie zaprezentowany sposób oparty o jedną tabelę (*Single Table*) zawierającą implementację atrybutów wszystkich encji (OSOBA, NAUCZYCIEL i STUDENT).

8. Ustawić we właściwościach encji OSOBA:

- *Subtree generation: Single Table*,
- *Apply to model: Relational_1*,
- *References: Identifying*,

a następnie uruchomić *Engineer to Relational Model*. Zmienić nazwę wygenerowanego modelu relacyjnego na *osoba_Single_Table*.

9. W tabeli KONTO zmienić nazwę kolumny *osoba_id_pracownika* na *id_pracownika* (w razie potrzeby).
10. Przeanalizować powstały model relacyjny. Zwrócić uwagę na połączenie struktur encji OSOBA, NAUCZYCIEL i STUDENT w jedną tabelę.



Dodatkowo warto zauważyć, że obowiązkowość atrybutów występujących w encjach NAUCZYCIEL i STUDENT (*nr_indeksu* i *tytul_naukowy*) została usunięta, co ma swoje logiczne uzasadnienie. W każdym wierszu tabeli OSOBA musi wystąpić *null* albo w kolumnie *nr_indeksu* albo *tytul_naukowy*. Może to być jedna z metod rozróżniania

pracowników uczelni. Można to zrealizować wybierając dla tabeli OSOBA (w modelu relacyjnym) poprzez *Properties / Table Level Constraints* i tworząc własną regułę walidacji:

Name: OSOBA_CH
Validation Rule:

*(nr_indeksu is null and tytul_naukowy is not null)
or
(nr_indeksu is not null and tytul_naukowy is null)*

Po wygenerowaniu skryptu można zauważać sposób implementacji tej reguły w postaci zdania SQL

```
alter table osoba
    add constraint osoba_ch check
        ((nr_indeksu is null and tytul_naukowy is not null)
        or
        (nr_indeksu is not null and tytul_naukowy is null));
```

umożliwiającą kontrolę poprawności wprowadzanych i modyfikowanych danych.

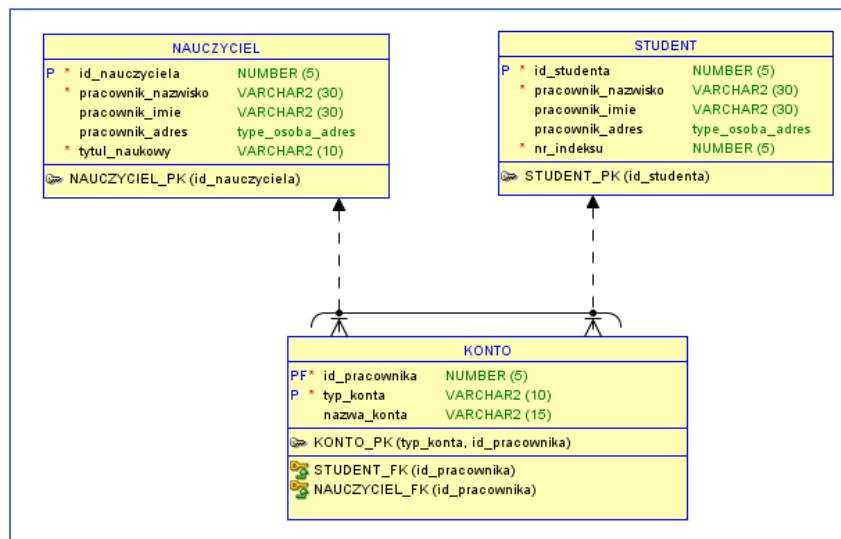
Skuteczność tego wariantu można przetestować realizując zdania SQL zawarte w dołączonym do materiałów skrypcie *osoba_inherit.sql*.

Inne warianty generowania modelu relacyjnego umieszczone w Oracle Data Modeler nie będą prezentowane w tym materiale.

Projektant lub programista może innymi metodami gwarantować rozróżnialność danej osoby w tworzonej aplikacji, na przykład wprowadzając w tabeli OSOBA status osoby w postaci kolumny z dopuszczalnymi wartościami określającymi nauczyciela i studenta.

Innym sposobem może być utworzenie dwóch oddzielnych tabel NAUCZYCIEL i STUDENT (bez tabeli OSOBA), dla których generowanie kolejnych wartości klucza głównego odbywa się poprzez jedną wspólną sekwencję, czyli definicje kolumn kluczy głównych mogą wyglądać tak:

```
id_nauczyciela numer(5) default seq_osoba.nextval primary key
id_studenta      numer(5) default seq_osoba.nextval primary key
```



W tym przypadku właściwość klucza obcego w tabeli KONTO (*F id_pracownika*) nie może być zagwarantowana na poziomie SQL w prosty sposób¹⁰, ale można to zrobić wspomagając się językiem PL/SQL, na przykład opracowując procedurę *pr_insert_konto*:

```
create or replace procedure pr_insert_konto
    (v_nr_pracownika numer,
     v_typ_konta varchar2 default 'mail',
     v_nazwa_konta varchar2) as
begin
    null;
end;
```

, która będzie dokonywała niezbędnych walidacji, aby programowo zapewnić spójność bazy danych czyli nie dopuścić do założenia konta nauczycielowi lub studentowi nie figurującemu w odpowiedniej ewidencji.

¹⁰ Przykład rozwiązania można znaleźć pod adresem <http://db-oriented.com/2016/03/11/implementing-arc-relationships-with-virtual-columns/> lub <https://stackoverflow.com/questions/61624669/how-do-you-recreate-the-arc-relationship-in-the-oracle-database-modeler-in-oracl>

	<i>Bazy Danych laboratorium</i>	Laboratorium BD8
--	-------------------------------------	-----------------------------

Zagadnienie: Programowanie serwera bazodanowego Oracle przy pomocy języka PL/SQL

I. Język PL/SQL - informacje podstawowe

PL/SQL jest skrótem od "Procedural Language extension to SQL" (proceduralne rozszerzenie języka SQL) i jest standardowym językiem programowania dostępnym w relacyjnych bazach danych Oracle.

Udostępnia blokową strukturę wykonywalnych modułów kodu, dzięki czemu konserwacja kodu jest znacznie łatwiejsza. Do podstawowych obiektów PL/SQL należą:

- zmienne, stałe i typy,
- struktury sterujące, takie jak instrukcje warunkowe i pętle,
- moduły wielokrotnego użytku, takie jak funkcje, procedury, wyzwalacze i pakiety.

Do zalet języka PL/SQL należą:

- integrowanie konstrukcji proceduralnych z językiem SQL - język SQL nie posiada konstrukcji warunkowych ani pętli w pełnym tego słowa znaczeniu, więc przy pomocy PL/SQL jest możliwa realizacja bardziej skomplikowanych algorytmów,
- zwiększenie wydajności serwera, gdyż obiekty PL/SQL znajdują się bezpośrednio na serwerze bazodanowym i tam są wywoływane i wykonywane,
- obiekty programowe są na trwałe zapamiętyane na serwerze bazodanowym i mogą być wielokrotnie wykonywane,
- umożliwia obsługę wyjątków (exception) występujących w trakcie wykonywania kodu,
- oprogramowanie utworzone w PL/SQL może być przenoszone między schematami (kontami), jak również między serwerami Oracle różnych wersji.

II. Typy obiektów PL/SQL i ich budowa

Do podstawowych bloków języka PL/SQL należą:

- bloki anonimowe,
- procedury,
- funkcje,
- wyzwalacze,
- pakiety programowe.

Bloki anonimowe

Są nietrwałymi obiektami programowymi czyli nie są zapamiętyane w schemacie, mogą być zapamiętyane w plikach tworzących skrypty wdrożeniowe (na przykład łącznie ze zdaniami SQL tworzącymi obiekty bazodanowe). Struktura takiego bloku wygląda tak:

```
DECLARE
    zmienne, kursory, wyjątki zdefiniowane przez użytkownika;
BEGIN
    zdania SQL - select, insert, update, delete;
    podstawowe instrukcje PL/SQL;
    wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
EXCEPTION
    instrukcje wykonywane w przypadku wystąpienia wyjątku (błędu);
END;
```

Przykład bloków anonimowych:

```
begin
    dbms_output.put_line ('PL/SQL');
end;
```

Sekcje DECLARE i EXCEPTION są opcjonalne. Obowiązkowe są instrukcje BEGIN i END stanowiące obszar dynamiczny kodu. Obszar ten nie może być pusty (*begin end;*), ale może zawierać *null* w swoim ciele (*begin null end;*).

Powyższy blok wyprowadza napis *PL/SQL* w środowisku SQL Developer lub SQL Plus.

Inny blok:

```
declare
    v_liczba_zawodnikow integer;
begin
    select count ( * ) into v_liczba_zawodnikow
    from bd3_zawodnicy;
    dbms_output.put_line ('Ewidencja zawiera ' || v_liczba_zawodnikow || ' zawodników');
end;
```

jest przykładem zanurzenia zdania SQL w kod programowy.

Procedury

Są trwałymi obiektami programowymi i ich rolą jest wykonanie pewnych czynności w bazie, na przykład utworzenie raportu analitycznego, wprowadzenie nowego wiersza do tabeli lub skasowanie wierszy spełniających pewne kryterium. Mogą być sparametryzowane lub też nie.

Budowa procedury:

```
CREATE OR REPLACE PROCEDURE nazwa_procedury
    ( lista parametrów wejściowych i wyjściowych )
AS
    zmienne, kursory, wyjątki zdefiniowane przez użytkownika;
BEGIN
    zdania SQL - select, insert, update, delete;
    podstawowe instrukcje PL/SQL;
    wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
EXCEPTION
    instrukcje wykonywane w przypadku wystąpienia wyjątku (błędu);
END;
```

Jak widać obszar dynamiczny między BEGIN i END jest dokładnie taki sam jak w bloku anonimowym oraz brak jest sekcji DECLARE.

```
create or replace procedure pr_insert_nowy_klub
    ( v_nr_klubu number,
      v_nazwa_klubu varchar2)
as
begin
    insert into bd3_kluby ( nr_klubu, nazwa_klubu )
    values ( v_nr_klubu, v_nazwa_klubu );
    commit;
end;
```

Należy zauważyc, że definicje parametrów formalnych procedury nie zawierają precyzji czyli występuje *number*, a nie *number (3)*. Analogicznie jest dla typu *varchar2*.

Deklaracji zmiennych lokalnych dokonuje się w obszarze między słowami kluczowymi *as* i *begin*.

Procedura uruchamiana jest przez swoją nazwę, na przykład:

```
declare
    v_nr_klubu integer default seq_nr_klubu.nextval;
    v_nazwa_klubu varchar2 ( 40 );
begin
    v_nazwa_klubu := 'Nowy Klub';
    pr_insert_nowy_klub ( v_nr_klubu, v_nazwa_klubu );
end;
```

Funkcje

Są trwałymi obiektami programowymi i ich podstawową rolą jest sprawdzanie stanu bazy danych oraz wykonywanie obliczeń lub znajdowanie w bazie danych konkretnych danych, na przykład sprawdzenie czy w ewidencji znajduje się zawodnik o określonych parametrami funkcji argumentach, obliczenie liczby punktów zdobytych przez danego zawodnika czy znalezienie w ewidencji najstarszego zawodnika. Mogą być sparametryzowane lub też nie.

Budowa funkcji:

```
CREATE OR REPLACE FUNCTION nazwa_funkcji
    ( lista parametrów wejściowych )
RETURN typ_funkcji
AS
    zmienne, kursorы, wyjątki zdefiniowane przez użytkownika;
BEGIN
    zdania SQL - select, insert, update, delete;
    podstawowe instrukcje PL/SQL;
    wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
RETURN zmienna_odpowiedniego_typu;
EXCEPTION
    instrukcje wykonywane w przypadku wystąpienia wyjątku ( błędu );
END;
```

Funkcja musi być określonego typu zdefiniowanego w nagłówku definicji (*RETURN ...*) oraz w ciele funkcji musi, co najmniej raz, wystąpić instrukcja *RETURN* wraz ze zwracaną wartością.

Przykład funkcji:

```
create or replace function fn_najstarszy_zawodnik_klubu
    ( v_nr_klubu number )
return varchar2
as
    v_nazwisko varchar2 ( 40 );
begin
    select nazwisko into v_nazwisko from bd3_zawodnicy
    where data_urodzenia = ( select min ( data_urodzenia )
                                from bd3_zawodnicy
                                where nr_klubu = v_nr_klubu );
return v_nazwisko;
end;
```

Funkcja zwraca wartość znakową i dlatego musi być typu *varchar2*.

Sposoby wywołania funkcji:

Pierwszy: w zdaniach SQL na tych samych zasadach jak funkcje standardowe języka SQL¹.

Zdanie:

```
select nr_klubu, fn_najstarszy_zawodnik_klubu ( nr_klubu ) "Nazwisko"
from bd3_kluby;
```

utworzy zbiór wyników:

NR_KLUBU	Nazwisko
1	Cisłak
2	Borowski
3	Werthein
4	Mańkowski
5	(null)
6	Dembiński
7	Kowalczyk

.....

Drugi: w bloku PL/SQL:

```
declare
    v_nr_klubu integer default :Nr_Klubu;
    v_nazwisko varchar2 ( 40 );
begin
    v_nazwisko := fn_najstarszy_zawodnik_klubu ( v_nr_klubu );
    dbms_output.put_line ( 'Najstarszym zawodnikiem w klubie ' || v_nr_klubu
                           || ' jest ' || v_nazwisko );
end;
```

Uruchomienie tego kodu z argumentem :Nr_Klubu = 1 da wynik:

```
Najstarszym zawodnikiem w klubie 1 jest Cisłak
```

Funkcja musi stać po prawej stronie instrukcji podstawienia i musi być takiego samego typu jak zmienna stojąca po lewej stronie.

Uwagi:

1. Bloki anonimowe nie są trwałym obiektem bazodanowym i ich "czas życia" jest zgodny z czasem aktywności sesji. Mogą znajdować się w historii pracy w SQL Developer lub być zapamiętanymi w plikach zewnętrznych.
2. Funkcje i procedury usuwane są ze schematu zdaniami:

```
drop procedure nazwa_procedury;
drop function nazwa_funkcji;
```

¹ Język SQL nie posiada typu logicznego (boolean), dlatego nie jest możliwe uruchamianie w ten sposób funkcji logicznych czyli funkcji zwracających wartość boolean.

3. Przy definiowaniu zmiennych lokalnych lub argumentów funkcji i procedur należy przestrzegać zasady, aby ich nazwy nie były tożsame z nazwami kolumn w tabelach, gdyż może to prowadzić do otrzymywania niezamierzonych wyników.

Rozważmy przykład:

```
declare
    nr_klubu integer;
    v_nazwa_klubu varchar2 ( 40 );
begin
    nr_klubu := 1;
    select nazwa_klubu into v_nazwa_klubu from bd3_kluby
    where nr_klubu = nr_klubu;
    dbms_output.put_line (v_nazwa_klubu);
end;
```

Próba uruchomienia powyższego kodu zakończy się błędem:

```
Error report -
ORA-01422: dokładne pobranie zwraca większą liczbę wierszy niż zamówiono
```

Warunek zawarty we frazie `where...` zdania `select...` jest warunkiem typu `1 = 1`, gdyż nazwy kolumn tabeli mają priorytet nad nazwami zmiennych.

III. Definiowanie zmiennych i nazw obiektów programowych

Konwencja nadawania nazw jest następująca:

- muszą zaczynać się od liter,
- mogą składać się z liter i cyfr,
- mogą zawierać znaki specjalne, takie jak:
 `$` , `_` , `#`
- ich długość nie może przekraczać 30 znaków,
- nie mogą być słowami zarezerwowanymi (np. `select` czy `begin`).

Przyjęło się, że nazwy zmiennych lub argumentów procedur i funkcji poprzedza się prefixem "v_" (v od *variable* - zmienna), np: `v_nazwisko`, `v_nr_klubu` tak, aby ustrzec się konfliktu opisanego wyżej.

Można też spotkać konwencję:

- `n_nr_klubu` dla zmiennych numerycznych,
- `c_nazwa_klubu` lub `v_nazwa_klubu` dla zmiennych znakowych,
- `d_data_urodzenia` dla zmiennych typu data.

Inicjowanie zmiennych może odbywać się na trzy sposoby.

Pierwszy polega na inicjowaniu zmiennej konkretną wartością w momencie jej deklarowania, drugi w obszarze dynamicznym instrukcją podstawienia i trzeci poprzez użycie zmiennej wiążącej zarówno w sekcji `DECLARE`, jak i w obszarze dynamicznym.

```
DECLARE
    v_data_urodzenia date;
    v_nr_klubu number( 3 ) := 1;
    c_napis constant varchar2 ( 50 ) default 'Ewidencja zawodników';
    v_nazwa_klubu varchar2 ( 40 ) := :Nazwa_klubu;
BEGIN
    v_nazwa_klubu := 'Nowy_Klub';
    v_nr_klubu := :Nr_klubu;
    .....
END;
```

Warto zauważyć, że zmienna niezainicjowana ma nieokreśloną wartość czyli *null*, jak w powyższym przykładzie zmienna *v_data_urodzenia*.

Zmienna *c_napis* ma atrybut *constant* (stała) czyli w przetwarzaniu nie może stać z lewej strony instrukcji podstawienia (nie może być zmieniona jej wartość).

Można inicjować również parametry formalne procedur i funkcji. Na przykład:

```
create or replace procedure pr_insert_nowy_klub
    ( v_nr_klubu number default seq_nr_klubu.nextval,
      v_nazwa_klubu varchar2)
....
```

lub

```
create or replace function fn_liczba_zawodnikow
    ( v_plec varchar2 := 'M' )
....
```

Sposób wywołania takiego kodu może uwzględniać wartości domyślne lub nie.

Na przykład dla powyżej zdefiniowanej procedury wywołanie jej w notacji pozycyjnej wygląda tak:

```
begin
    pr_insert_nowy_klub ( 50, 'Nowy Klub' );
end;
```

a dla notacji nominalnej:

```
begin
    pr_insert_nowy_klub ( v_nazwa_klubu => 'Nowy Klub' ); -- numer klubu określi sekwencja
end;
```

Typy zmiennych

Podstawowymi typami zmiennych języka PL/SQL są:

- *number* dla zmiennych numerycznych,
- *varchar2* dla zmiennych znakowych,
- *date* i *timestamp* dla dat i czasu,
- *boolean* dla zmiennych logicznych.

Lista możliwych typów jest dłuższa. Poza typami skalarnymi istnieje możliwość definiowania typów złożonych, takich jak rekordy czy tablice, jak również typy dotyczące dużych obiektów (takich jak zdjęcia, pliki multimedialne). Zagadnienia te nie będą w tym dokumencie omawiane.

Dodatkowo w stosunku do typów numerycznych można stosować znane z innych języków programowania typy, takie jak *integer*, *real*, czy *float*.

Atrybut %TYPE

Atrybut ten służy do deklarowania zmiennej zgodnie z definicją kolumny w tabeli lub inną zadeklarowaną zmienną.

```
DECLARE
    v_nazwisko          bd3_zawodnicy.nazwisko%type;
    v_liczba_kobiet     integer;
    v_liczba_mezczyzn   v_liczba_kobiet%type;
```

Można go również używać do definiowania argumentów procedur i funkcji, na przykład:

```
create or replace function fn_liczba_zawodnikow
    ( v_plec bd3_zawodnicy.plec%type := 'M' )
```

Ten sposób deklarowania zmiennych zwiększa stabilność oprogramowania, gdyż typy zmiennych są dynamicznie dziedziczone w momencie uruchamiania kodu. Oprogramowanie jest mniej narażone na błędy fatalne, które mogą wystąpić na skutek zmiany struktur tabel bazodanowych.

Zmienne logiczne nie występują w języku SQL (nie ma kolumny typu *boolean*), a w języku PL/SQL - tak. Można im przypisać wyłącznie wartości TRUE, FALSE i NULL.

Do zwracania wartości logicznych można używać wyrażeń arytmetycznych, znakowych i dat zbudowanych w postaci zdań logicznych.

```
DECLARE
    v_old number := 400;
    v_new number := 500;
    flag boolean;
BEGIN
    flag := v_old < v_new; -- wartość zmiennej flag zostanie ustawiona na TRUE
END;
```

Typ *boolean* może być również użyty do definiowania funkcji logicznych. Przykładowo chcemy opracować funkcję decyzyjną, która będzie odpowiadała na pytanie: "Czy zawodnik o zadanym numerze startował w jakichkolwiek zawodach?".

Kod takiej funkcji może wyglądać tak:

```
create or replace function fn_czy_zawodnik_startował
    (v_nr_zawodnika bd3_wyniki.nr_zawodnika%type)
return boolean
as
if_exists integer;
how_many integer;
begin
    select count(*) into if_exists
        from bd3_zawodnicy
        where nr_zawodnika = v_nr_zawodnika;
    select count(*) into how_many
        from bd3_wyniki
        where nr_zawodnika = v_nr_zawodnika;

    case
        when if_exists = 0 then return null;
        when how_many = 0 then return false;
        when how_many > 0 then return true;
    end case;
end;
```

Uruchomienie takiej funkcji w zdaniu SQL nie jest możliwe z uwagi na brak typu *boolean* w implementacji tego języka. Natomiast jej wywołanie w kodzie PL/SQL typu procedura czy blok anonimowy jest możliwe i spełnia taką samą rolę jak zmienna logiczna użyta na przykład w funkcji sterującej *if* lub *case*.

Zdanie:

```
select fn_czy_zawodnik_startował (v_nr_zawodnika => 333) from dual;
```

zgłosi błąd wykonania:

```
SQL Error: ORA-00902: niepoprawny typ danych
```

Natomiast można tej funkcji użyć do sterowania przetwarzaniem:

```

declare
v_nr_zaw bd3_wyniki.nr_zawodnika%type := :Nr_Zawodnika;
begin
if fn_czy_zawodnik_startował (v_nr_zawodnika => v_nr_zaw) then
    /* gdy TRUE */
    dbms_output.put_line ('Zawodnik o numerze ''||v_nr_zaw||' startował w sezonie');

elsif not fn_czy_zawodnik_startował (v_nr_zawodnika => v_nr_zaw) then
    /* gdy FALSE */
    dbms_output.put_line ('Zawodnik o numerze ''||v_nr_zaw||' nie startował w sezonie');

else          /* gdy NULL */
    dbms_output.put_line ('Zawodnika o numerze ''||v_nr_zaw||' brak w ewidencji');
end if;
end;

Zawodnik o numerze 50 nie startował w sezonie
Zawodnik o numerze 333 startował w sezonie
Zawodnika o numerze 1333 brak w ewidencji

```

IV. Podstawowe konstrukcje programowe w PL/SQL

Język PL/SQL jest językiem strukturalnym zawierającym szereg, znanych z innych języków programowania, konstrukcji programowych, takich jak instrukcje warunkowe i pętle. I z tego względu nie będą one dokładnie omawiane w tym rozdziale. Zostaną podane tylko przykłady ich zastosowania w odniesieniu do baz danych. Więcej informacji można będzie znaleźć w dokumentacji Oracle lub na rekomendowanym wcześniej serwisie [Tech on the Net](#).

Instrukcja warunkowa if ... then else end if;

Steruje wykonywaniem się odpowiednich partii kodu programowego w zależności od badanego warunku.

Na przykład:

```

.....
select count ( * ) into v_liczba_zawodnikow
from bd3_zawodnicy
where plec = 'K' and nr_klubu = v_nr_klubu;

if v_liczba_zawodnikow = 0 then
    dbms_output.put_line ( 'W klubie o numerze ' || v_nr_klubu || ' brak jest zawodników ');
else
    dbms_output.put_line ( 'W klubie o numerze ' || v_nr_klubu || ' jest '
                           || v_liczba_zawodnikow || ' zawodników ' );
end if;
.....

```

Innymi odmianami tej konstrukcji są:

```

if ..... then ..... end if;
if ..... then ..... elsif ..... then ..... end if;
if ..... then ..... elsif ..... then ..... else ..... end if;

```

Należy zwrócić uwagę na zachowanie się instrukcji warunkowej przy obsłudze wartości *null*.

Przykład pierwszy:

```
.....
x := 5;
y := NULL;
...
IF x <> y THEN -- wartością nie jest TRUE, lecz NULL
..... -- ta część algorytmu nie zostanie wykonana
END IF;
```

Przykład drugi:

```
.....
a := NULL;
b := NULL;
...
IF a = b THEN -- wartością nie jest TRUE, lecz NULL
..... -- ta część algorytmu nie zostanie wykonana
ELSE
..... -- ta część algorytmu zostanie wykonana
END IF;
```

Wyrażenie i instrukcja case

W języku PL/SQL konstrukcja *case* występuje w dwóch postaciach, jako wyrażenie *case* i jako instrukcja *case*.

Wyrażenie *case* ma dwie postacie.

Pierwsza postać:

```
declare
identyfikator number ( 3 ) := upper( :zmienna_wejsciowa);
v_zmienna_wyjsciowa varchar2 (40);

begin
v_zmienna_wyjsciowa :=
    case identyfikator
        when 200 then ' Wspaniale' -- identyfikator badany jest na równość z wartościami
        when 100 then ' Bardzo dobrze '
        when 50 then ' Wystarczająco '
        else ' Brak danych ' -- jeśli brak frazy else i żaden z powyższych
                           -- warunków nie jest spełniony - case zwraca null
    end;
dbms_output.put_line ( v_zmienna_wyjsciowa );
end;
```

Druga postać:

```
declare
v_suma_pkt integer;
v_medal varchar2(40);
```

```

begin

select sum( nvl(punkty_globalne, 0) ) into v_suma_pkt
from bd3_wyniki
where nr_zawodnika = :Podaj_numer_zawodnika; -- suma punktów dla zawodnika niestartującego ani
                                              razu jest null

v_medal :=
  case
    when v_suma_pkt = 0      then ' Brak medalu ' -- zawodnik startował i nie zdobył
                                  punktów
    when v_suma_pkt < 100   then ' Medal pocieszenia '
    when v_suma_pkt < 130   then ' Brązowy medal '
    when v_suma_pkt < 160   then ' Srebrny medal '
    when v_suma_pkt >= 160 then ' Złoty medal '
    else ' Zawodnik nie startował w sezonie '     -- zawodnik ani razu nie startował
  end;

dbms_output.put_line( v_medal );
end;

```

W każdej frazie `when` może wystąpić dowolne zdanie logiczne (niekoniecznie zawierające tę samą zmienną). Pierwsze zdanie logiczne dające wartość TRUE kończy przeszukiwanie konstrukcji `case`. Jeśli żaden warunek nie jest spełniony zwracana jest wartość frazy `else`, a jeśli jej nie ma - wartość `null`.

Natomiast instrukcja `case` realizuje różne warianty algorytmu (wykonuje instrukcje) w zależności od wartości zmiennej stanowiącej selector:

```

.....
case v_action
  when ' DEL '    then delete bd3_kluby_kopia;
  when ' COUNT '  then select count( * ) into v_liczba from bd3_zawodnicy;
  when ' SUM '    then .....
.....
end case;
.....

```

Pętle sterujące

Pętle to konstrukcje, które wielokrotnie powtarzają wykonywanie instrukcji lub sekwencji instrukcji. Są trzy rodzaje pętli:

- pętla podstawowa
- pętla `while`
- pętla `for`

Pętla podstawowa `loop`:

```

loop
  {przetwarzane instrukcje PL/SQL i SQL}
  exit {warunek wyjścia z pętli}
end loop;

```

Warunek wyjścia z pętli musi być tak oprogramowany, aby dawał możliwość wyjścia z niej po skończonej liczbie iteracji.

W przypadku braku frazy `exit` lub nie spełnienia powyższego postulatu pętla będzie nieskończona.

Przykład:

```
.....
counter integer default 1;
max_iteracja integer := :ile_razy;

begin
.....
loop
    insert into tabela_wartosci (numer_wiersza)
    values (counter * 100);
    exit when counter = max_iteracja;
    counter := counter + 1;
end loop;
.....
end;
```

Tak zbudowana pętla wykona się co najmniej raz.

Pętla **while**:

```
while {warunek wyjścia z pętli}
loop
    {przetwarzane instrukcje PL/SQL i SQL}
end loop;
```

Przykład:

```
.....
counter integer default 1;
max_iteracja integer := :ile_razy;

begin
.....
while counter <= max_iteracja loop
    insert into tabela_wartosci (numer_wiersza)
    values (counter * 100);
    counter := counter + 1;
end loop;
.....
end;
```

Warunek wyjścia z pętli umieszczony jest przy wejściu do kolejnej iteracji i dlatego tak zbudowana pętla może nie wykonać się ani razu.

W pętlach *loop* i *while* nie jest znana liczba iteracji przed rozpoczęciem jej wykonywania, gdyż warunek wyjścia może być dynamicznie zmieniany w trakcie iteracji.

Pętla **for**:

```
for counter in low_value..high_value
loop
    {przetwarzane instrukcje PL/SQL i SQL}
end loop;
```

Przykład:

```
.....
max_iteracja integer = :ile_razy;
begin
.....
for counter in 1 .. max_iteracja loop
    insert into tabela_wartosci (numer_wiersza)
    values (counter * 100);
end loop;
.....
```

Liczba iteracji określona jest przy wejściu do pętli na podstawie zmiennych low_value i high_value. Licznik pętli (counter) nie musi być definiowany i nie może być miejscem docelowym żadnej instrukcji podstawienia.

V. Wykonywanie zdań języka SQL w PL/SQL

Zdanie select:

Budowa zdania *select* zanurzonego w PL/SQL rozszerzona jest o frazę *into* określającą zmienną lub zmienne, w których będą umieszczone dane pobrane z bazy.

```
select kolumna1, kolumna2, .....
  into zmienna1, zmienna2, .....
  from {tabela lub tabele}
  where .....
  having ....
```

Przykład:

```
create or replace procedure pr_liczba_zawodnikow
    (v_nr_klubu bd3_zawodnicy.nr_klubu%type)
as
v_nazwa_klubu bd3_kluby.nazwa_klubu%type;
v_liczosc integer;

begin
    select nazwa_klubu, count ( * )
        into v_nazwa_klubu, v_liczosc
    from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
    where z.nr_klubu = v_nr_klubu
    group by nazwa_klubu;

    dbms_output.put_line ('Klub'|| v_nazwa_klubu ||' ma zarejestrowanych ' ||
                           v_liczosc || ' zawodników' );
end;
```

Wywołanie tej procedury:

```
begin
    pr_liczba_zawodnikow ( 3 );
end;
```

da wynik:

```
| Klub KB Gymnasion Warszawa ma zarejestrowanych 104 zawodników
```

Zmienne użyte we frazie *into* są typu skalarnego (liczby, napisy czy daty) i dlatego zdanie *select* musi być tak zbudowane, aby zwracało wartości skalarne a nie zbiory. Innymi słowy warunek we frazie *where* musi określać dokładnie jeden wiersz.

Aby się o tym przekonać wystarczy w ciele procedury zmienić filtr we frazie *where* na

```
.....  
where z.nr_klubu > v_nr_klubu  
.....
```

i ponownie ją wywołać. Otrzymamy błąd uruchomienia:

```
Error report -  
ORA-01422: dokładne pobranie zwraca większą liczbę wierszy niż zamówiono
```

Zdanie *select* zwraca więcej niż jeden wiersz i nie można tych zbiorów umieścić w zmiennych skalarnych.

Podobnie, jeśli zdanie *select* nie zwraca żadnego wiersza (na przykład podany numer klubu nie istnieje), wtedy także zwracany jest błąd:

```
begin  
    pr_liczba_zawodnikow ( 0 );  
end;  
  
Error report -  
ORA-01403: nie znaleziono danych
```

Efektem wystąpienia tych błędów jest przerwanie się wykonania kodu w aplikacji, co jest zjawiskiem niepożądany. Aby temu przeciwdziałać należy te błędy wychwytywać i odpowiednio obsługiwać. Realizuje się to w sekcji EXCEPTION kodu PL/SQL, co nie będzie omówione w tym materiale.

Zdania DML (*insert, update, delete*):

W celu zmiany stanu bazy danych poprzez wprowadzanie danych do tabel, ich modyfikacje lub kasowanie używane są zdania SQL: *insert*, *update* i *delete*. Mogą być używane bezpośrednio w skryptach SQL lub w kodzie PL/SQL. Istotne jest to, że w przeciwieństwie do zdania *select* (które w PL/SQL zmienia swoją postać) zdania DML mają taką samą budowę.

Na przykład chcąc wprowadzić dane o nowych zawodach do tabeli BD3_ZAWODY można napisać procedurę:

```
create or replace procedure pr_insert_nowe_zawody  
    (v_nr_zawodow numeric,  
     v_nazwa_zawodow varchar2,  
     v_data_zawodow date,  
     v_podtytuł varchar2 default null)  
as  
begin  
    insert into bd3_zawody (nr_zawodow, nazwa_zawodow, data_zawodow, podtytuł)  
    values (v_nr_zawodow, v_nazwa_zawodow, v_data_zawodow, v_podtytuł);  
    commit;  
end;
```

Wywołując ją blokiem anonimowym:

```
begin
    pr_insert_nowe_zawody (5, 'Puchar Warszawy', '2018/11/10');
    pr_insert_nowe_zawody (6, 'Puchar Warszawy', '2018/11/11', 'Bieg Niepodległości');
end;
/
select * from bd3_zawody;
```

otrzymujemy wynik:

NR_ZAWODOW	NAZWA_ZAWODOW	DATA_ZAW	PODYTUL
5	Puchar Warszawy	18/11/10	
6	Puchar Warszawy	18/11/11	Bieg Niepodległości
1	Cztery Pory Roku - Jesień	10/10/22	
2	Cztery Pory Roku - Zima	11/02/19	
3	Cztery Pory Roku - Wiosna	11/05/21	
4	Cztery Pory Roku - Lato	11/09/03	im.S.Dankowskiego
6 rows selected.			

Uwagi:

1. Znak "/" między kodem PL/SQL i zdaniem SQL nakazuje systemowi zarządzania bazą danych przełączyć się z kontekstu PL/SQL na kontekst SQL lub odwrotnie.
2. Należy zwrócić uwagę na atrybut formalny `v_podtytuł` w specyfikacji procedury. Ponieważ została ustalona wartość domyślna (w tym przypadku `null`) wywołanie tej procedury może mieć dwojaką postać: z wprowadzeniem podtytułu lub nie, co widać w powyższym bloku uruchamiającym tę procedurę dwa razy. Zapoznając się ze strukturą tabeli BD3_ZAWODY można stwierdzić, że kolumna `PODYTUL` może zawierać wartości `null` i dlatego taka specyfikacja procedury jest możliwa. Zmiana specyfikacji nagłówka tej procedury:

```
create or replace procedure pr_insert_nowe_zawody
(v_nr_zawodow number,
 v_nazwa_zawodow varchar2 default null,
 v_data_zawodow date,
 v_podtytuł varchar2 default null)
```

oraz próba uruchomienia jej:

```
begin
    pr_insert_nowe_zawody (v_nr_zawodow => 7,
                           v_data_zawodow => '2018/11/20');
end;
```

zakończy się błędem, gdyż kolumna `NAZWA_ZAWODOW` w tabeli nie może przyjmować wartości `null`.

3. Usuwając wartości domyślne w specyfikacji procedury:

```
create or replace procedure pr_insert_nowe_zawody
(v_nr_zawodow number,
 v_nazwa_zawodow varchar2,
 v_data_zawodow date,
 v_podtytuł varchar2 )
```

i wywołując ją tak jak poprzednio:

```
begin
    pr_insert_nowe_zawody (7, 'Puchar Warszawy', '2018/11/20');
end;
```

otrzymamy błąd niepoprawnej liczby argumentów, mimo, że kolumna PODTYTUL może przyjmować wartości *null*.

Jak wcześniej było napisane wykonanie zdanie *select* w kodzie PL/SQL może spowodować przerwane wykonywania kodu PL/SQL (a więc aplikacji), w przypadku gdy zbiór wynikowy zdania *select* jest zbiorem pustym.

Weźmy pod uwagę zdanie *select* wykonane interaktywnie:

```
select nazwisko, imie
from bd3_zawodnicy
where nr_zawodnika = 1000;
```

Zakładając, że nie ma zawodnika o takim numerze otrzymamy odpowiedź:

no rows selected

lub

 NAZWISKO	 IMIE
--------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------

w zależności od sposobu uruchomienia, ale brak jest błędu.

Jeśli tak skonstruowane zdanie umieszczone zostanie w kodzie PL/SQL:

```
declare
v_nazwisko bd3_zawodnicy.nazwisko%type;
v_imie bd3_zawodnicy.imie%type;
begin
    select nazwisko, imie
    into v_nazwisko, v_imie
    from bd3_zawodnicy
    where nr_zawodnika = 1000;
    dbms_output.put_line ( 'Zdanie zostało wykonane' );
end;
```

i wykonane, to kod "załamie się" w zdaniu *select* i dalsze przetwarzanie (wyświetlenie komunikatu) zostanie przerwane.

W przypadku użycia zdań DML w kodzie PL/SQL sytuacja jest inna.

Założymy, że chcemy skasować wszystkie wyniki zawodów o określonym numerze, którego brak w ewidencji zawodów.

Wykonując blok:

```
begin
    delete bd3_wyniki
    where nr_zawodow = 10;
    dbms_output.put_line ( 'Żądanie kasowania wyników zostało wykonane' );
end;
```

i zakładając, że danych o zawodach o numerze 10 nie ma otrzymamy wynik:

Żądanie kasowania wyników zostało wykonane

, co oznacza, że zdanie DML (w tym przypadku *delete*) nie zostało przerwane, mimo, że żaden wiersz w tabeli BD3_WYNIKI nie spełniał kryterium kasowania.

Podobnie jest ze zdaniem *update*. Jeśli żaden wiersz nie spełnia kryteriów modyfikacji stan bazy się nie zmieni, a działanie aplikacji nie zostanie przerwane.

Do analizy liczby wierszy poddanych działaniu zdań *delete* lub *update* pomocnym może być niejawnym kursem SQL%ROWCOUNT. Z każdym zdaniem DML wykonywanym przez serwer skojarzony jest indywidualny kurSOR zarządzany przez kod PL/SQL. Jego wartość ustawiana jest po każdym wykonaniu zdania *select*, *insert*, *update* i *delete* i określa liczbę wierszy objętą ostatnio wykonanym zdaniem SQL.

Na przykład poniższy kod:

```
begin
    delete bd3_wyniki
    where nr_zawodow = 10;
    dbms_output.put_line ( 'Skasowano '|| sql%rowcount || ' wierszy' );
end;
```

po uruchomieniu wyświetli komunikat:

Skasowano 0 wierszy

a zmieniając numer zawodów na istniejący, np. 1 wynik będzie inny:

Skasowano 274 wierszy

Należy mieć świadomość, że z tabeli wyników zostały usunięte wiersze, ale nie w sposób trwałym. Chcąc wycofać skutki działania tego kodu należy wycofać transakcję zdaniem SQL *rollback* według scenariusza:

```
select count( * ) from bd3_wyniki
where nr_zawodow =1;                                -- sprawdzenie liczby wierszy przed wycofaniem
                                                       -- transakcji (zbiór jest pusty)

rollback;                                            -- wycofanie transakcji

select count( * ) from bd3_wyniki
where nr_zawodow =1;                                -- sprawdzenie liczby wierszy po wycofaniu
                                                       -- transakcji (zbiór liczy 274 wiersze)
```

VI. Zastosowanie kursorów w PL/SQL

Kursorem w języku PL/SQL określana jest technika polegająca na wykonaniu zdania *select* i utworzeniu zbioru wynikowego, a następnie na możliwości czytania i przetwarzania wiersz po wierszu tego zbioru.

W ujęciu klasycznym użycie kursora obejmuje następujące fazy:

- zdefiniowanie kursora zdaniem *select*,
- otwarcie kursora instrukcją *open*,
- czytanie kursora instrukcją *fetch*,
- sprawdzanie końca zbioru wynikowego funkcją *%notfound*,
- zamknięcie kursora instrukcją *close*.



Przykład zastosowania kurSORA niesparametryzowanego:

```

declare
v_nazwisko bd3_zawodnicy.nazwisko%type;
v_data_urodzenia bd3_zawodnicy.data_urodzenia%type;
cursor cur_zawodnik is
    select nazwisko, data_urodzenia
    from bd3_zawodnicy
    order by data_urodzenia desc;

begin
open cur_zawodnik;
loop
    fetch cur_zawodnik into v_nazwisko, v_data_urodzenia;
    exit when cur_zawodnik%notfound;
    dbms_output.put_line ( v_nazwisko || ' ' ||
                           to_char ( v_data_urodzenia, 'DD.MM.YYYY' ) );
end loop;
close cur_zawodnik;
end;

```

Procedura drukowania w powyższym przykładzie symbolizuje przetwarzanie otrzymanych kursorem danych. Dane te można przetwarzać na różny sposób w zależności od potrzeb biznesowych.

Tak zdefiniowany kurSOR ma charakter statyczny, gdyż za każdym razem przy jego uruchomieniu zbiór wynikowy będzie taki sam.

Definicję kurSORA można sparametryzować przy pomocy zmiennej wiązania albo poprzez atrybut procedury.

Na przykład:

```
create or replace procedure pr_zawodnicy_punkty
    (v_nr_klubu bd3_zawodnicy.nr_klubu %type)
as
v_zawodnik varchar2(100);
v_suma integer;
cursor cur_agregat is
    select nazwisko || ' ' || imie zawodnik, sum ( nvl(punkty_globalne, 0 ) ) suma
    from bd3_zawodnicy z join bd3_wyniki w
        on z.nr_zawodnika = w.nr_zawodnika
    where z.nr_klubu = v_nr_klubu
    group by nazwisko || ' ' || imie
    order by suma desc;

begin
open cur_agregat;
dbms_output.put_line ('Osiągnięcia zawodników');
dbms_output.put_line ('=====');
dbms_output.put_line ( CHR ( 10 ));
loop
    fetch cur_agregat into v_zawodnik, v_suma;
    exit when cur_agregat%notfound;
    dbms_output.put_line (v_zawodnik || ' zdobył '|| v_suma || ' pkt');
end loop;
dbms_output.put_line ('=====');
close cur_agregat;
end;
```

Definicja kursora jest dynamiczna, gdyż numer klubu określany jest w momencie uruchomienia procedury, na przykład blokiem:

```
begin
    pr_zawodnicy_punkty ( :nr_klubu );
end;
```

lub w ciele innej procedury.

Kolejnym sposobem przetwarzania danych przy pomocy kursora jest *kursorowa pętla for*, budowa której oparta jest o klasyczną pętlę *for*. Występuje w dwóch wariantach.

Przykład pierwszego wariantu:

```
declare
cursor cur_klub is
    select nr_klubu, nazwa_klubu
    from bd3_kluby
    order by nazwa_klubu;
begin
    for tmpRec in cur_klub
    loop
        dbms_output.put_line ( 'Klub ' || tmpRec.nazwa_klubu ||
                               ' ma numer ' || tmpRec.nr_klubu);
    end loop;
end;
```

Tak skonstruowana metoda użycia nie wymaga otwierania, zamykania ani badania końca zbioru kursora. Należy w sekcji DECLARE zdefiniować sparametryzowany lub nie kursor, a w części dynamicznej zastosować pętlę *for*.

Zmienna *tmpRec* nie wymaga wcześniejszego definiowania i jej struktura (skalarne lub rekordowa) jest taka, aby można było w niej umieścić to co zwraca kursor przy jednym obrocie pętli, czyli w powyższym przykładzie numer i nazwę klubu.

Przykład drugiego wariantu:

```
begin
  for tmpRec in ( select nazwisko, sum ( punkty_globalne ) suma
    from bd3_zawodnicy z join bd3_wyniki w
    on z.nr_zawodnika = w.nr_zawodnika
    group by nazwisko
    having sum ( punkty_globalne ) >= :Próg_punktowy
    order by suma desc )
  loop
    dbms_output.put_line ( 'Zawodnik ' || tmpRec.nazwisko ||
      ' zdobył ' || tmpRec.suma || ' pkt');
  end loop;
end;
```

Ta metoda dodatkowo nie wymaga definiowania kursora. Zdanie *select* jest użyte bezpośrednio w konstrukcji pętli *for*.

Zadania do samodzielnego wykonania

- Opracować procedurę wyświetlającą datę zawodów oraz liczbę startujących w nich zawodników. Argumentem wejściowym jest data zawodów. W przypadku, gdy w zadanym dniu nie było zawodów należy wyświetlić komunikat o braku zawodów w tym dniu.
- Opracować procedurę, która wpisuje do tabeli klubów nowy klub. Argumentem wejściowym ma być nazwa klubu. Wartość numeru klubu obliczać przy pomocy funkcji *max(nr_klubu) + 1*. Procedura ma sprawdzać, czy taka nazwa klubu już istnieje w tabeli. Jeśli tak należy wyświetlić stosowny komunikat.
- Opracować procedurę, która wyświetla zadaną liczbę najlepszych zawodników danych zawodach w następujący sposób: Nazwisko, Nazwa Klubu, czas_min:czas_sek. Argumentami wejściowymi ma być numer zawodów oraz liczba wyświetlanych zawodników.
- Opracować procedurę, która na wejściu posiada atrybuty: numer klubu oraz Action mogący przyjmować następujące wartości: COUNT, SUMA, AVG w postaci napisu.
W zależności od tego atrybutu funkcja będzie zwracała:
 - COUNT - liczbę aktywnych zawodników czyli tych, którzy startowali w sezonie chociaż raz,
 - SUMA - sumę zdobytych punktów przez zawodników tego klubu,
 - AVG - średnią wieku zawodników tego klubu w rozbiciu na płeć.
 Do tego celu opracować odpowiednią perspektywę lub perspektywy i jej / ich używać w ciele funkcji. Użyć tej funkcji w kodzie programowym do wyświetlania wyników (wedle uznania).
- Napisać procedurę, która przyjmuje jako dane wejściowe numer istniejącego klubu oraz nowy numer tego klubu. Zadaniem jej jest zmiana numeru klubu na nowy w ewidencji klubów, jak również zmiana numeru klubu w ewidencji zawodników.
- Opracować funkcję logiczną, która stwierdza czy zawodnik o zadany numerze startował w zawodach o określonym numerze. W przypadku nieistnienia zadanej numeru zawodnika lub numeru zawodów funkcja powinna zwracać *null*.

7. Słownik kategorii wiekowych (BD3_KATEGORIE) zawiera przedziały czasowe w postaci roczników, które decydują o przynależności każdego zawodnika do odpowiedniej kategorii:

NR_KATEGORII	NAZWA_KATEGORII	DOLNY_PROG	GORNY_PROG
1 I		1998	2017
2 II		1988	1997
3 III		1978	1987
4 IV		1968	1977
5 V		1958	1967
.....			
11 XI		1	1927
20 K-I		1998	2017
21 K-II		1988	1997
22 K-III		1978	1987
23 K-IV		1968	1977
.....			

, przy czym nr_kategorii = 1 oznacza najmłodszą kategorię męską,
= 11 oznacza najstarszą kategorię męską,
= 20 oznacza najmłodszą kategorię żeńską,
= 30 oznacza najstarszą kategorię żeńską.

Po zakończeniu rocznego cyklu biegowego czyli na początku każdego roku kalendarzowego musi nastąpić modyfikacja przynależności zawodników do odpowiednich kategorii wiekowych. Odbywało się to według następującego scenariusza:

- Modyfikacja tabeli BD3_KATEGORIE polegająca na przesunięciu okna czasowego dla każdej kategorii o jeden rok, czyli przykładowo na początku roku 2018 dla kategorii o numerze 1 nowe wartości okna czasowego powinny wynosić 1999 .. 2018, a dla kategorii o numerze 2 – 1989 .. 1998 itd. Dla najstarszych kategorii obojga płci wartości w kolumnie dolny_prog mają charakter symboliczny i nie muszą podlegać modyfikacji.
- Dla wszystkich zawodników (ale osobno dla mężczyzn i osobno dla kobiet) następowało powtórne obliczenie przynależności do odpowiednich kategorii wiekowych przy pomocy zdań SQL:

```
update bd3_zawodnicy
set nr_kategorii = (select nr_kategorii
                      from bd3_kategorie
                      where extract (year from data_urodzenia)
                            between dolny_prog and gorny_prog
                      and nr_kategorii between 1 and 11 )
where plec = 'M';
```

```
update bd3_zawodnicy
set nr_kategorii = (select nr_kategorii
                      from bd3_kategorie
                      where extract (year from data_urodzenia)
                            between dolny_prog and gorny_prog
                      and nr_kategorii between 20 and 30 )
where plec = 'K';
```

- Zatwierdzenie transakcji zdaniem commit,

Należy opracować procedurę, która zautomatyzuje ten proces oraz uniemożliwi powtórną modyfikację tabel BD3_KATEGORIE i BD3_ZAWODNICY w tym samym roku kalendarzowym. Ta druga funkcjonalność może być zapewniona przez sprawdzenie wartości w kolumnie *gorny_prog* dla kategorii o numerze 1. W tym polu zawsze będzie znajdował się bieżący rok, jeśli nastąpiło już przeliczenie na jego początek.

Dodatkowo można zautomatyzować wykonanie się tej procedury poprzez opracowanie zadania, które może być uruchamiane raz w roku (np. każdego 1 stycznia o godzinie 8:00). Tworzenie zadań (*jobs*) zostało opisane w Laboratorium BD9.

	<i>Bazy Danych laboratorium</i>	Laboratorium BD9
--	-------------------------------------	-----------------------------

Zagadnienie: Konstruowanie i zastosowanie automatów programowych w PL/SQL

Na potrzeby zajęć zostanie wykorzystany model bazy danych opisany i zaimplementowany w ramach Laboratorium BD7.

I. Czynności wstępne

Przy pomocy zdań SQL należy zmodyfikować zaimplementowany model sprzedaży zawierający trzy tabele z prefixem BD4 wykonując poniższe czynności:

1. W tabeli BD4_PRODUKT dodać kolumnę ILOSC_W_MAGAZYNIE, która będzie przechowywała ilość danego produktu w magazynie. Ustawić jako domyślną wartość 0.
2. Dla już wprowadzonych produktów wprowadzić konkretną wartość do tej kolumny - dla uproszczenia można wszystkim produktom nadać tę samą wartość.
3. W tabeli BD4_RACHUNEK dodać kolumnę STATUS_RACHUNKU i określić dla niej przy pomocy definicji CHECK dopuszczalne wartości: OTWARTE, ZAMKNIĘTE i ANULOWANE obrazujące stan realizacji danego zamówienia.
4. Usunąć wszystkie rachunki z tabeli BD4_RACHUNEK oraz utworzyć na nowo sekwencję seq_rachunek.

II. Wyzwalacze (triggers) jako automaty bazodanowe

Wyzwalacze są blokami programowymi, które są uruchamiane automatycznie (niejawnie) w wyniku zajścia określonego zdarzenia. Najczęściej stosowane są w sytuacji wystąpienia w kodzie programowym zdania DML (*insert, update, delete*) do pewnej tabeli. W przypadku, gdy dla tej tabeli skonstruowany jest odpowiedni wyzwalacz – jest on uruchamiany przed lub po zdaniu DML.

Jego rolą może być uzupełnienie zdania DML, które wywołało wyzwalacz, realizacja zdania DML w innej tabeli lub walidacja danych czyli sprawdzenie poprawności danych wejściowych w procedurze lub bezpośrednio w zdaniu DML.

Są dwa typy wyzwalaczy: wyzwalacze zdania i wyzwalacze wiersza.

Wyzwalacz zdania (*for statement*) jest uruchamiany raz dla zdania DML (zdarzenia wyzwalającego) bez względu na liczbę wierszy objętych działaniem tego DML, nawet jeśli liczba wierszy wynosi zero.

Wyzwalacz wiersza (*for each row*) jest uruchamiany dla każdego wiersza objętego działaniem zdarzenia wyzwalającego. Jeśli żaden wiersz nie jest objęty jego działaniem, nie dochodzi w ogóle do uruchomienia wyzwalacza.

Przykładowo, jeśli dla jakiejś tabeli został zdefiniowany wyzwalacz typu *for statement* reagujący na zdanie *update* do tej tabeli, to w momencie uruchomienia zdania:

```
update tabela
    set kolumna = nowa_wartosc
  where kolumna_inna <= 3;
```

scenariusz będzie taki:

1. *update* dla kolumna_inna = 1,
2. *update* dla kolumna_inna = 2,

3. *update* dla kolumna_inna = 3,
4. wywołanie (odpalenie - fired) wyzwalacza. -- jednorazowe uruchomienie

Jeśli dla tej samej tabeli został zdefiniowany wyzwalacz typu *for each row* reagujący na zdanie *update* do tej tabeli, to w momencie uruchomienia tego samego zdania scenariusz będzie inny:

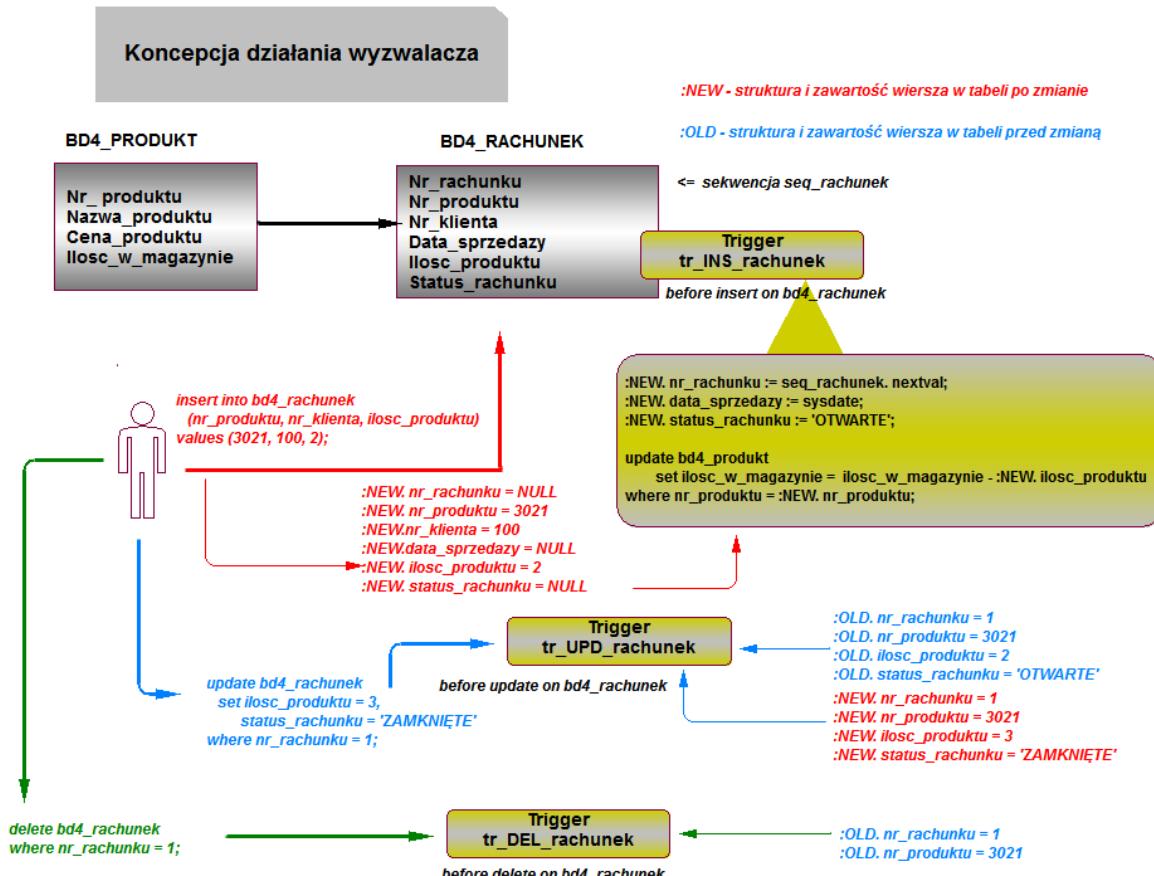
1. *update* dla kolumna_inna = 1,
2. wywołanie wyzwalacza,
3. *update* dla kolumna_inna = 2,
4. wywołanie wyzwalacza,
5. *update* dla kolumna_inna = 3,
6. wywołanie wyzwalacza.

Moment uruchomienia wyzwalacza zależy od klauzuli BEFORE lub AFTER.

BEFORE – treść wyzwalacza jest wykonywana przed zdaniem DML,

AFTER – treść wyzwalacza jest wykonywana po zdaniu DML.

Przy pomocy poniższego diagramu zostanie omówiona koncepcja działania wyzwalacza bazodanowego.



Z wyzwalaczami związane są dwie struktury: :OLD i :NEW. Ich budowa jest tożsama ze strukturą tabeli, z którą dany wyzwalacz jest związany, czyli jeśli tabela ma strukturę:

nr_produktu,
nazwa_produktu,
cena_produktu,
ilosc_w_magazynie,
....

to obie powyższe struktury mają taką samą budowę i można się do elementów tych struktur odwoływać:

:NEW.nr_produktu :OLD.nr_produktu
 :NEW.nazwa_produktu :OLD.nazwa_produktu

i tak dalej.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *insert*

Wprowadzenie nowej pozycji do tabeli BD4_RACHUNEK wiąże się z określeniem: numeru rachunku, numeru klienta, numeru produktu i ilości tego produktu, daty złożenia zamówienia oraz statusu rachunku. Niektóre z tych danych mogą być ustawiane automatycznie. Należą do nich: data złożenia zamówienia (*sysdate*), status rachunku (w momencie składania zamówienia status jest OTWARTE) oraz numer rachunku (klucz główny tabeli może być wyznaczany przez sekwencję). Od wprowadzającego nowe zamówienie wymagane będzie podanie tylko numeru produktu i jego ilości oraz numeru klienta, który to zamówienie składa. Pozostałe wielkości zostaną ustawione w wyzwalaczu.

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_ins_rachunek
before insert on bd4_rachunek
for each row
begin
  :new.nr_rachunku := seq_rachunek.nextval;
  :new.data_sprzedazy := sysdate;
  :new.status_rachunku := 'OTWARTE';
end;
```

, a zdanie DML wyzwalające go:

```
insert into bd4_rachunek (nr_produktu, ilosc_produktu, nr_klienta)
values (3021, 2, 100);
```

Struktury :NEW i :OLD przed wykonaniem się zdania *insert* będą wyglądały następująco:

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	Null	Null	2	3021	100	Null
:OLD	Null	Null	Null	Null	Null	Null

Działanie wyzwalacza spowoduje, że struktura :NEW się zmieni:

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	1	sysdate	2	3021	100	OTWARTE
:OLD	Null	Null	Null	Null	Null	Null

I tak skonstruowane zdanie *insert* zostanie wykonanie.

Rolą tego wyzwalacza było uzupełnienie zdania *insert* o standardowe wartości kolumn. Uzupełnienie to nastąpiło przed wykonaniem się zdania *insert* z racji określenia klauzuli BEFORE (najpierw wyzwalacz, a potem *insert*).

Drugim zadaniem wyzwalacza może być działanie w innej tabeli. W omawianym przykładzie w momencie złożenia zamówienia musi nastąpić rezerwacja żądanej ilości produktu czyli modyfikacja jego ilości w magazynie (w tabeli BD4_PRODUKT).

Można zatem rozszerzyć kod wyzwalacza o tę funkcjonalność:

```
.....  
update bd4_produkt  
    set ilosc_w_magazynie = ilosc_w_magazynie - :new.ilosc_produktu  
  where nr_produktu = :new.nr_produktu;
```

Przetestować działanie tak skonstruowanego wyzwalacza można według scenariusza:

1. Odczytać z tabeli produktów ilość danego produktu w magazynie,
2. Złożyć zamówienie na ten produkt w ilości nie przekraczającej stanu magazynu,
3. Odczytać zawartość ewidencji zamówień,
4. Sprawdzić stan zamówienia produktu w magazynie.

Uwagi:

1. W powyższym fragmencie kodu wyzwalacza struktura :NEW jest strukturą tabeli BD4_RACHUNEK, a nie tabeli BD4_PRODUKT.
2. W kodzie wyzwalacza nie można używać zdań DML do tej samej tabeli, na przykład w powyższym wyzwalaczu zabronione jest użycie zdania *update BD4_RACHUNEK lub delete BD4_RACHUNEK*, natomiast można używać zdania *select ... from BD4_RACHUNEK*.
3. W wyzwalaczu nie można używać zdań *commit* i *rollback*.
4. W przypadku konieczności zdefiniowania wewnątrz wyzwalacza zmiennych należy użyć sekcji **DECLARE**:

```
.....  
for each row  
declare  
    [deklaracje zmiennych tak jak w bloku anonimowym PL/SQL]  
begin  
    .....  
end;
```

5. Chcąc zdefiniować wyzwalacz zdania należy opuścić klauzulę *for each row*.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *update*

W przypadku konieczności modyfikacji szczegółów zamówienia w tabeli BD4_RACHUNEK należy wykonać zdanie:

```
update bd4_rachunek  
    set ilosc_produktu = 3 -- zmiana tylko ilości zamówionego produktu  
  where nr_rachunku = 1;
```

lub

```
update bd4_rachunek  
    set ilosc_produktu = 2,  
        nr_produktu = 4268      -- zmiana asortymentu i ilości  
  where nr_rachunku = 1;
```

Zmiana warunków zamówienia pociąga za sobą zmianę ilości towarów w magazynie, a więc w tabeli BD4_PRODUKT.

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_upd_rachunek
before update on bd4_rachunek
for each row
begin
    update bd4_produkt
        set ilosc_w_magazynie = ilosc_w_magazynie - :new.ilosc_produktu
        + :old.ilosc_produktu
        where nr_produktu = :new.nr_produktu;
end;
```

Struktury :NEW i :OLD przed wykonaniem się zdania *update* (dotyczącego tylko zmiany ilości zamówionego produktu) będą wyglądały następująco:

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	1	2021/01/26	3	3021	100	OTWARTE
:OLD	1	2021/10/26	2	3021	100	OTWARTE

I tak skonstruowane zdanie *update* zmieni szczegóły zamówienia, a wyzwalacz zmieni ilość produktu w magazynie.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *delete*

W przypadku konieczności skasowania złożonego zamówienia (wiersza w tabeli BD4_RACHUNEK) można postępować dwójako. Albo skasować to zamówienie bezpowrotnie zmieniając odpowiednio stan magazynowy produktu w tabeli BD4_PRODUKT albo nie kasować zamówienia, tylko zmienić jego status na ANULOWANE i zmienić stan magazynowy produktu.

Poniżej zostanie zaprezentowane rozwiązanie dla pierwszego wariantu.

W przypadku konieczności skasowania zamówienia w tabeli BD4_RACHUNEK należy wykonać zdanie:

```
delete bd4_rachunek
where nr_rachunku = 1;
```

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_del_rachunek
before delete on bd4_rachunek
for each row
begin
    update bd4_produkt
        set ilosc_w_magazynie = ilosc_w_magazynie + :old.ilosc_produktu
        where nr_produktu = :old.nr_produktu;
end;
```

Struktury :NEW i :OLD przed wykonaniem się zdania *delete* będą wyglądały następująco:

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	Null	Null	Null	Null	Null	Null
:OLD	1	2021/01/26	3	3021	100	OTWARTE

I tak skonstruowane zdanie *delete* usunie złożone zamówienie, a wyzwalacz "zwróci" nie sprzedany a zarezerwowany produkt do magazynu.

Uwagi:

1. W przypadku wyzwalacza reagującego na zdanie *update* możliwe jest wskazanie kolumny lub kolumn, zmiana których spowoduje uruchomienie zaprojektowanego wyzwalacza.

Na przykład wyzwalacz:

```
create or replace trigger tr_upd_rachunek
before update of ilosc_produktu on bd4_rachunek
for each row
```

.....
będzie reagował na zdanie:

```
update bd4_rachunek
set ilosc_produktu = 3
where nr_rachunku = 1;
```

, ale nie będzie reagował na zdanie:

```
update bd4_rachunek
set nr_produktu = 4265
where nr_rachunku = 1;
```

2. Można łączyć w jednym wyzwalaczu akcje podejmowane w przypadku wykonywania różnych zdań DML, na przykład:

```
create or replace trigger tr_rachunek
before insert or delete or update on bd4_rachunek
```

.....
W takim przypadku mogą być pomocne predykaty warunkowe: *INSERTING*, *UPDATING* i *DELETING* do budowania kodu wyzwalacza:

```
if INSERTING then .....end if;
if UPDATING then .... .end if;
if UPDATING ('ilosc_produktu') then.....end if;
if UPDATING ('nr_produktu') then ..... end if;
if DELETING then ..... end if;
```

3. Można budować wyzwalacz w oparciu o perspektywę. Na przykład:

Załóżmy, że istnieje perspektywa *bd4_rachunek_produkt* zawierająca poniższe kolumny:

Nr_rachunku, Data_sprzedazy, Nazwa_produktu, Cena_produktu, Nazwisko_klienta

W oparciu o tę perspektywę można rejestrować nowe zamówienie, nawet w przypadku, gdy nazwisko klienta nie figuruje w ewidencji klientów lub brak jest w ewidencji produktu o podanej nazwie.

Nagłówek wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_ins_rachunek_view
instead of insert on bd4_rachunek_produkt
```

.....

Realizując zdanie:

```
insert into bd4_rachunek_produkt (Nr_rachunku, Data_sprzedazy,
Nazwa_produktu,..... )
values (.....);
```

należy poprzez wyzwalacz zapewnić, aby w tabeli BD4_PRODUKT znalazła się produkt występujący w powyższym zdaniu, jeśli go tam jeszcze nie ma. Analogicznie należy postępować w przypadku nowego klienta.

Kod takiego wyzwalacza zawierać może takie zdania *insert* i/lub *update* do różnych tabel zapewniające skuteczne wprowadzenie nowego zamówienia do tabeli BD4_RACHUNEK.

Walidacja danych przy pomocy wyzwalacza

Walidacja wprowadzanych danych do tabeli czyli kontrola ich poprawności może być realizowana przy pomocy wyzwalacza.

Przykładowo chcąc wpisać nowy produkt do tabeli BD4_PRODUKT można kontrolować, czy produkt o zadanej nazwie już występuje w tabeli. Ponieważ nazwa produktu nie jest kluczem głównym, więc należy dokonać dodatkowej programowej kontroli. Można to zrealizować poprzez wyzwalacz reagujący na zdanie *insert* do tabeli BD4_PRODUKT. Jego zadaniem będzie stwierdzenie, czy podana nazwa produktu już w tabeli istnieje i jeśli tak dać komunikat typu *raise* czyli przerwanie wykonywania się zdania *insert*.

Procedura wpisująca nowy produkt do tabeli BD4_PRODUKT może wyglądać tak:

```
create or replace procedure pr_insert_produkt
    (v_nr_produktu numeric,
     v_nazwa_produktu varchar2,
     v_cena_produktu numeric,
     v_rok_produkcji numeric default null,
     v_ranking number default 5,
     v_ilosc_w_magazynie numeric) AS
begin
    insert into bd4_produkt
        values (v_nr_produktu, v_nazwa_produktu, v_cena_produktu,
                v_rok_produkcji, v_ranking, v_ilosc_w_magazynie);

    dbms_output.put_line ( ' Dalsze przetwarzanie...' );
end;
```

W ciele procedury instrukcja *dbms_output.put_line* symbolizuje dalsze (ewentualne) przetwarzanie danych.

Można opracować wyzwalacz reagujący na zdanie *insert* do tabeli produktów o postaci:

```
create or replace trigger tr_ins_produkt
before insert on bd4_produkt
for each row
declare
    v_ile integer;
begin
    select count( * ) into v_ile
    from bd4_produkt
    where nazwa_produktu = :new. nazwa_produktu;

    if v_ile = 1 then
        raise_application_error (-20001,
                               'Produkt'||:new. nazwa_produktu||' już jest zarejestrowany');
    end if;
end;
```

Zakładając, że w tabeli BD4_PRODUKT nie ma produktu o nazwie Canon 6D Mark II i uruchamiając powyższą procedurę, na przykład blokiem:

```

begin
  pr_insert_produkt (4021, 'Canon 6D Mark II', 6599, 2017, 6, 15);
end;

```

w panelu Dbms Output zostanie wyświetlony komunikat *Dalsze przetwarzanie...* pochodzący z procedury, co oznacza, że nowy produkt został wpisany do tabeli:

NR_PRODUKTU	NAZWA_PRODUKTU	CENA_PRODUKTU	ROK_PRODUKCJI	RANKING	ILOSC_W_MAGAZYNIE
4021	Canon 6D Mark II	6599	2017	6	15
3021	Canon 6D Body	4450	2016	8	10
3055	Obiektyw EF 50mm	519,99	2012	(null)	10
4265	Obiektyw EF 24-105mm	669	2015	6	10
4268	Obiektyw EF 70-200mm	3649	2014	(null)	10

, a wyzwalacz nie wykazał powtórzonej nazwy.

Jeśli ponownie będziemy chcieli wprowadzić do tabeli produkt o tej samej nazwie i nowym numerze produktu:

```

begin
  pr_insert_produkt (5021, 'Canon 6D Mark II', 6399, 2017, 6, 5);
end;

```

otrzymamy komunikat zgłoszony przez *raise_application_error* w wyzwalaczu:

```

Error report -
ORA-20001: Produkt Canon 6D Mark II już jest zarejestrowany

```

, a w panelu Dbms Output nie pojawi się komunikat *Dalsze przetwarzanie...*, co oznacza, że działanie procedury zostało przerwane czyli **cała** (!!!) procedura się nie wykonała i ten sam produkt nie został wpisany do tabeli.

Procedura *raise_application_error* pozwala projektantowi tworzyć własne komunikaty błędu i umieszczać je w kodzie PL/SQL.

Pierwszy argument może przyjmować wartości z zakresu -20000 do -20999, a drugi zawiera komunikat, który ma być przekazany w wyniku wystąpienia błędu (wyjątku - exception).

Jeśli taki kod PL/SQL będzie umieszczony w aplikacji utworzonej w jakimś języku programowania, to sposób dalszej obsługi takiego komunikatu może być różny. Temat ten nie będzie omawiany w tym materiale.

Zadania do samodzielnego wykonania

- Zmodyfikować tabelę BD4_KLIENT dodając do niej kolumnę STATUS_Klienta. Kolumna ta będzie określała status klienta na podstawie sumarycznej wielkości przeprowadzanych transakcji według zasady:

Jeśli suma zakupów przekroczy wartość HIGH klient otrzymuje status BARDZO WAŻNY, jeśli ta suma jest w zakresie LOW i HIGH to klient otrzymuje status WAŻNY, w przeciwnym przypadku nie ma żadnego statusu.

Opracować kod PL/SQL (blok anonimowy lub procedurę), który jednorazowo na podstawie zawartości tabeli BD4_RACHUNEK dokona modyfikacji zawartości kolumny STATUS_Klienta.

Opracować wyzwalacz, który na bieżąco będzie dokonywał modyfikacji tej kolumny w momencie rejestrowania nowego zamówienia w tabeli BD4_RACHUNEK.

2. Opracować jeden zbiorczy wyzwalacz łączący w sobie omówione w materiale wyzwalacze zbudowane w oparciu o tabelę BD4_RACHUNEK i reagujący na zdania *insert*, *update* i *delete*. Wykorzystać predykaty warunkowe *INSERTING*, *UPDATING* i *DELETING*.
3. Utworzyć tabelę słownikową BD4_RABATY zawierającą wartości procentowe rabatów przyznawanych klientom o odpowiednim statusie przy składaniu kolejnych zamówień. Tabela może składać się z dwóch kolumn: nazwa_statusu (HIGH i LOW) oraz wartości progowych (np. 10% i 5%).
Utworzyć odpowiednią relację między tabelami BD4_RABATY i BD4_KLIENCI.
Zmodyfikować tabelę BD4_RACHUNEK dodając do niej dwie kolumny: RABAT oraz WARTOSC_RACHUNKU. Kolumna RABAT otrzymuje wartość zgodną ze statusem klienta, a wartość rachunku obliczana jest na podstawie ilości zamówionego produktu, ceny tego produktu i przydzielonego rabatu.
Zmodyfikować, opracowany w punkcie 2, wyzwalacz uwzględniający przyznawanie rabatu klientowi przy składaniu zamówień.

III. Zadania (jobs) jako automaty czasowe

Omawiane do tej pory automaty typu wyzwalacze działały w ten sposób, że były wyzwalane zdarzeniem jakim mogło być jedno ze zdań DML (*insert*, *update* czy *delete*).

Automaty czasowe zwane zadaniami (częściej "jobami") są wyzwalane czasem. Można na przykład określić zadanie polegające na wydruku raportu analitycznego pierwszego dnia każdego miesiąca o godzinie 06:16, czy też uruchamiać konkretną procedurę co dwie godziny.

Podstawowymi pojęciami używanymi w tej tematyce są: *schedule*, *program* i *job*.

Schedule (harmonogram) - terminarz wykonywania jobów. Można zrobić harmonogram który określa realizację pewnej czynności z określona częstotliwością i skojarzyć go z kilkoma jobami. Dzięki temu kilka jobów uruchamianych jest zgodnie z jednym harmonogramem i w przypadku konieczności jego zmiany robi się to w jednym miejscu. Odpowiada na pytanie: *Kiedy to ma się wykonać?*

Program – w nim można zdefiniować działania przy pomocy bloku anonimowego, procedury PL/SQL czy też pliku zewnętrznego z poziomu systemu operacyjnego. Można go skojarzyć z wieloma jobami i również w przypadku jego zmiany robi się to w jednym miejscu. Odpowiada na pytanie: *Co ma się wykonać?*

Job - łączy w sobie definicje harmonogramu i programu czyli odpowiada na pytanie: *Jaki program i zgodnie z jakim harmonogramem ma się wykonać?* Jako jedyny z tych obiektów ma charakter dynamiczny, to znaczy tylko uruchomienie joba zrealizuje zadanie. Harmonogramy i programy stanowią statyczne definicje.

Do zarządzania zadaniami (tworzenie zadań, uruchamianie ich oraz kasowanie) służy specjalny pakiet programowy Oracle o nazwie *dbms_scheduler*.

Poniżej zostaną przedstawione przykładowe sposoby definiowania elementów umożliwiających wykorzystanie jobów w realizacji zadań w bazie danych.

1. Tworzenie harmonogramów przy użyciu dbms_scheduler.create_schedule

```

begin
  -- codziennie od Monday do Friday o godzinie 22:00

  dbms_scheduler.create_schedule

    (schedule_name => 'INTERVAL_DAILY_2200',
     start_date=> trunc(sysdate)+18/24,      -- start dzisiaj o 18:00
     repeat_interval=>'freq=DAILY; byday=MON,TUE,WED,THU,FRI; byhour=22',
     comments=>'Uruchamiane (Mon-Fri) o 22:00'
    );
end;
-----
begin
  -- codziennie co godzinę

  dbms_scheduler.create_schedule

    (schedule_name => 'INTERVAL_EVERY_HOUR',
     start_date  => trunc ( sysdate ) + 18/24,      -- uaktywnienie nastąpi o godzinie 18:00
     repeat_interval => 'freq=HOURLY; interval=1',
     comments    => 'Uruchamiane codziennie co godzinę'
    );
  -- codziennie co 5 minut

  dbms_scheduler.create_schedule

    (schedule_name => 'INTERVAL_EVERY_5_MINUTES',
     start_date  => trunc ( sysdate +1 ) + 20/24/60,   -- uaktywnienie nastąpi o godzinie 00:20
     end_date    => trunc ( sysdate ) + 30,           -- następnego dnia
     repeat_interval => 'freq=MINUTELY; interval=5',
     comments    => 'Uruchamiane codziennie co 5 minut'
    );
  -- codziennie co minutę przez 30 dni

  dbms_scheduler.create_schedule

    (schedule_name => 'INTERVAL_EVERY_MINUTE',
     start_date  => trunc ( sysdate +1 ) + 20//24/60,
     end_date    => trunc ( sysdate ) + 30,
     repeat_interval => 'freq=MINUTELY; interval=1',
     comments    => 'Uruchamiane codziennie przez 30 dni co 1 minutę'
    );
  -- w każdą niedzielę o godzinie 18:00

  dbms_scheduler.create_schedule

    (schedule_name => 'INTERVAL_EVERY_SUN_1800',
     start_date=> trunc ( sysdate ) + 18/24,
     repeat_interval=> 'freq=DAILY; byday=SUN; byhour=18;',
     comments=>'Uruchamiane w niedzielę o godzinie 18'
    );
end;

```

2. Tworzenie programów przy użyciu dbms_scheduler.create_program

begin

-- wywołanie bloku anonimowego

```
dbms_scheduler.create_program
  (program_name => 'PROG_INIT_DRAWING_TABLE',
  program_action =>

    'BEGIN
      delete from DRAWING_TABLE;
    END; ',  

  program_type => 'plsql_block',
  comments => 'Inicjowanie tabeli DRAWING_TABLE',
);  

end;
```

begin

-- wywołanie wbudowanej procedury

```
dbms_scheduler.create_program
  (program_name=> 'PROG_DRAWING_ONE_VALUE',
  program_type=> 'stored_procedure',
  program_action=> 'pr_generation_drawing_value',
  comments=>'Losowanie pojedynczej wartości'  

);  

end;
```

begin

-- wywołanie pakietowej procedury

```
dbms_scheduler.create_program
  (program_name=> 'PROG_PERCENT_DRAWING_TABLE',
  program_type=> ' stored_procedure ',
  program_action=> 'pkg_drawing.pr_count_percent',
  comments=>'Obliczanie procentów w DRAWING_TABLE'  

);  

end;
```

3. Tworzenie zadania (job) przez połączenie harmonogramu z programem przy użyciu dbms_scheduler.create_job:

begin

-- połączenie harmonogramu z programem

```
dbms_scheduler.create_job
  (job_name => 'JÓB_DRAWING_ONE_VALUE',
  program_name=> 'PROG_DRAWING_ONE_VALUE',
  schedule_name=>'INTERVAL_EVERY_MINUTE',
  comments=>'Losowanie jednej wartości co 1 minutę');
```

```

dbms_scheduler.create_job
(job_name => 'JOB_PERCENT_DRAWING_TABLE',
program_name=> 'PROG_PERCENT_DRAWING_TABLE',
schedule_name=>'INTERVAL_EVERY_5_MINUTES',
comments=>'Obliczanie procentów w DRAWING_TABLE');

end;

```

4. Tworzenie zadania (job) bez definicji harmonogramu i programu przy użyciu dbms_scheduler.create_job:

Jedną z właściwości pakietów programowych jest możliwość przeciążania procedur w nich zawartych. Procedura pakietowa *create_job* jest taką właśnie procedurą. Dzięki temu można definiować zadania na różne sposoby.

Na przykład zatrzymać w jego definicji nazwę programu oraz szczegółowe parametry harmonogramu:

```

.....
dbms_scheduler.create_job
(job_name => 'JOB_DRAWING_ONE_VALUE',
program_name=> 'PROG_DRAWING_ONE_VALUE',
start_date => trunc ( sysdate ) + 18/24,
end_date => trunc ( sysdate ) + 30,
repeat_interval => 'freq=MINUTELY; interval=1',
comments=>'Losowanie jednej wartości co 1 minutę przez 30 dni');

.....

```

lub w ogóle nie definiować nazw programu i harmonogramu:

```

.....
dbms_scheduler.create_job
(job_name => 'JOB_PERCENT_DRAWING_TABLE',

program_type=> 'stored_procedure',
program_action=> 'pkg_drawing.pr_count_percent';

start_date => trunc ( sysdate ) + 18/24,
repeat_interval => 'freq=MINUTELY; interval=5'

comments=>'Obliczanie procentów w DRAWING_TABLE');

.....

```

Procedura *create_job* zawiera w sobie wszystkie możliwe argumenty formalne występujące w procedurach *create_program* i *create_schedule*.

5. Uruchamianie zadań (jobów):

```

begin
  dbms_scheduler.run_job ( 'JOB_DRAWING_ONE_VALUE' );
  dbms_scheduler.run_job ( 'JOB_PERCENT_DRAWING_TABLE' );
end;

```

6. Restart zadania:

W przypadku konieczności wznowienia działania zadania na skutek, na przykład, zmiany jego parametrów należy zadanie deaktywować i ponownie aktywować:

```
begin
    dbms_scheduler.disable ( 'JOB_DRAWING_ONE_VALUE' );
    dbms_scheduler.enable ( 'JOB_DRAWING_ONE_VALUE' );

    dbms_scheduler.disable ( 'JOB_PERCENT_DRAWING_TABLE' );
    dbms_scheduler.enable ( 'JOB_PERCENT_DRAWING_TABLE' );
end;
```

7. Metadane związane z zadaniami, programami i harmonogramami

Definicje wszystkich obiektów związanych z zadaniami można analizować poprzez wyświetlanie ich metadanych poniższymi zdaniami SQL:

```
select * from user_scheduler_jobs;
select * from user_scheduler_programs;
select * from user_scheduler_schedules;
```

8. Usuwanie definicji zadań, programów i harmonogramów

```
begin
    dbms_scheduler.drop_job ( 'JOB_DRAWING_ONE_VALUE' );
    dbms_scheduler.drop_job ( 'JOB_PERCENT_DRAWING_TABLE' );

    dbms_scheduler.drop_program ( 'PROG_DRAWING_ONE_VALUE' );
    dbms_scheduler.drop_program ( 'PROG_INIT_DRAWING_TABLE' );
    dbms_scheduler.drop_program ( 'PROG_PERCENT_DRAWING_TABLE' );

    dbms_scheduler.drop_schedule ( 'INTERVAL_DAILY_2200' );
    dbms_scheduler.drop_schedule ( 'INTERVAL_EVERY_5_MINUTES' );

end;
```

Uwagi:

- Istnieje możliwość zmiany zdefiniowanych wcześniej parametrów zadań, programów i harmonogramów przy pomocy procedury pakietowej `dbms_scheduler.set_attribute`, na przykład:

```
dbms_scheduler.set_attribute
(
    name => 'INTERVAL_EVERY_MINUTE',
    attribute => 'start_date',
    value => to_date ( '23.12.2020 12:30' , 'dd.mm.yyyy hh24:mi' )
);
lub
dbms_scheduler.set_attribute
(
    name => 'INTERVAL_EVERY_MINUTE',
    attribute => 'repeat_interval',
    value => 'freq=MINUTELY;interval=2'
);
```

- Istnieje również możliwość przekazywania poprzez programy i zadania argumentów do procedur poprzez te obiekty wykonywanych. Zagadnienie to nie będzie omawiane w tym materiale.

	<i>Bazy Danych laboratorium</i>	Laboratorium BD10
--	-------------------------------------	------------------------------

Zagadnienie: Tworzenie raportów w środowisku Oracle przy pomocy Jasper Reports

Jasper Reports jest opartym o Javę narzędziem umożliwiającym tworzenie zaawansowanych raportów, które można wyświetlać na ekranie, drukować na drukarce czy też eksportować do formatów takich jak PDF, HTML, MSEExcel, RTF, ODT, XML.

I. Konfiguracja środowiska Jasper Reports

1. Do opracowywania raportów zostanie użyte narzędzie *iReport Designer*. Jest to narzędzie służące do projektowania wzorców raportów dostępne na stronie:

<http://community.jaspersoft.com/project/ireport-designer>

lub w zasobach uczelnianych.

Rozpakowaną zawartość należy umieścić w dowolnym folderze swojego komputera:

Adres C:\iReport-5.6.0			
Nazwa	Rozmiar	Typ	Data modyfikacji
bin		Folder plików	2016-07-01 16:29
etc		Folder plików	2016-07-01 16:29
ide10		Folder plików	2016-07-01 16:29
ireport		Folder plików	2016-07-01 16:28
license-text-files		Folder plików	2016-07-01 16:28
nb6.5		Folder plików	2016-07-01 16:28
platform9		Folder plików	2016-07-01 16:28
Changelog	43 KB	Dokument tekstowy	2014-05-28 09:31
LICENSE_ireport	35 KB	Dokument tekstowy	2013-02-20 09:07
notice	4 KB	Dokument tekstowy	2013-02-20 09:07
readme	1 KB	Dokument tekstowy	2013-02-20 09:07
Third-Party-Notices	279 KB	Adobe Acrobat Docu...	2014-05-28 08:19

2. Oprogramowanie *iReport Designer* korzysta z bibliotek Java (jdk....) i dlatego należy zainstalować odpowiednią dla systemu operacyjnego wersję.¹ Przed pierwszym uruchomieniem *iReport Designer* należy zmodyfikować plik konfiguracyjny...iReport-5.6.0\etc\ireport.conf:

.....

default location of JDK/JRE, can be overridden by using --jdkhome <dir> switch
jdkhome="C:\Program Files\Java\jdk1.7.0_79"

W odkomentowanej linii jdkhome należy podać pełną ścieżkę do biblioteki jdk.

¹ Dla Windows XP odpowiednią wersją jest jdk1.6.0_xx, a dla Windows 7, 8 i 10 - jdk1.7.0_xx. Nowsze wersje bibliotek Javy, np. jdk1.8.0_xx mogą uniemożliwić prawidłowe funkcjonowanie oprogramowania iReport Designer.

3. Jeśli na komputerze jest zainstalowane oprogramowanie Oracle w postaci serwera bazodanowego lub klienta bazodanowego to odpowiedni sterownik znajduje się w *ORACLE_HOME*, np.:

Adres	C:\app\Administrator\product\11.2.0\client_1\jdbc\lib		
Nazwa	Rozmiar	Typ	Data modyfikacji
ojdbc5	1 950 KB	Executable Jar File	2010-02-23 22:09
ojdbc5_g	3 010 KB	Executable Jar File	2010-02-23 22:09
ojdbc5dms	2 374 KB	Executable Jar File	2010-02-23 22:09
ojdbc5dms_g	3 030 KB	Executable Jar File	2010-02-23 22:09
<u>ojdbc6</u>	2 062 KB	Executable Jar File	2010-02-23 22:09
ojdbc6_g	3 323 KB	Executable Jar File	2010-02-23 22:09
ojdbc6dms	2 594 KB	Executable Jar File	2010-02-23 22:09
ojdbc6dms_g	3 344 KB	Executable Jar File	2010-02-23 22:09
<u>ojdbc7</u>	3 613 KB	Executable Jar File	2016-07-21 18:52
simplefan	20 KB	Executable Jar File	2010-02-23 22:09

4. Sterowniki do bazy Oracle można również pobrać ze strony:

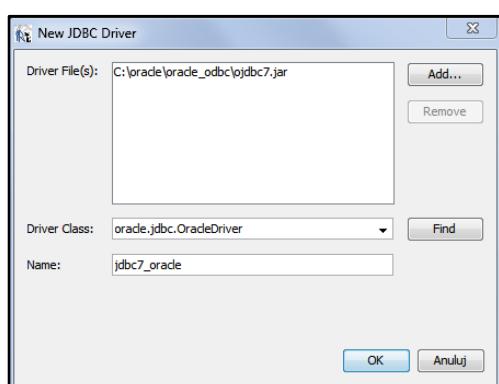
<http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html>

lub z zasobów uczelnianych (*Laboratorium BD10.zip*).

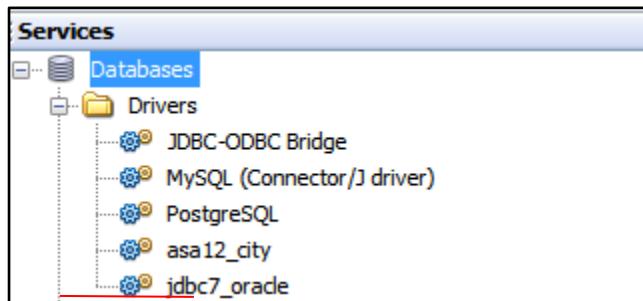
Pobrany sterownik można umieścić w dowolnym folderze komputera, np. w folderze ..\Report-5.6.0.

II. Definiowanie połączenia ze schematem na serwerze Oracle

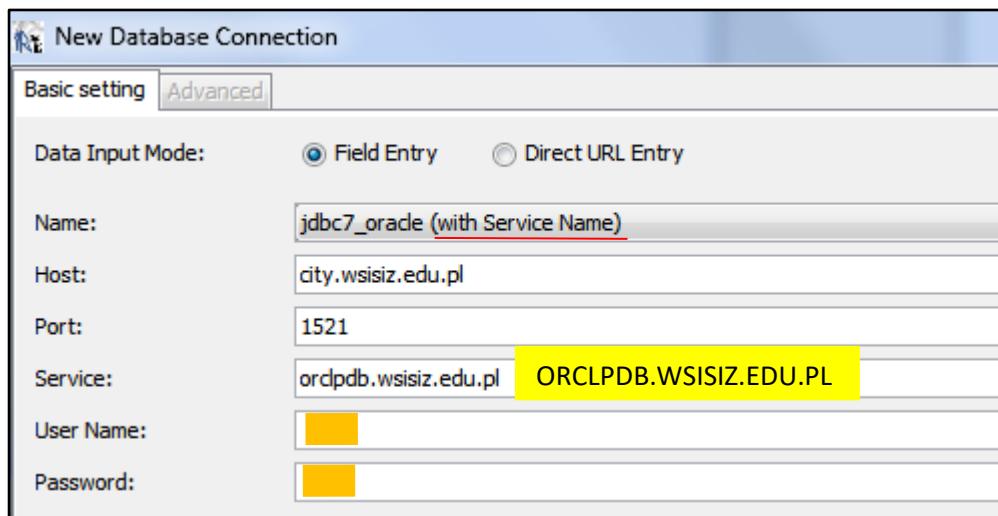
1. Po uruchomieniu *iReport Designer* należy z menu *Okno/Uslugi*, rozwinąć drzewko *Databases/Drivers*, kliknąć prawym przyciskiem myszy na *Drivers* i wybrać *New driver*. Zarejestrować, poprzez przyciski *Add.../Find*, driver *jdbc*, nadając mu własną nazwę, np.:



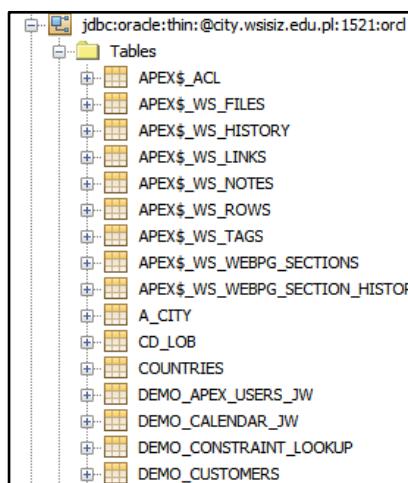
2. Wybierając prawym przyciskiem myszy zarejestrowany driver:



można przystąpić do definiowania połączenia ze schematem bazodanowym (*Connect Using*):

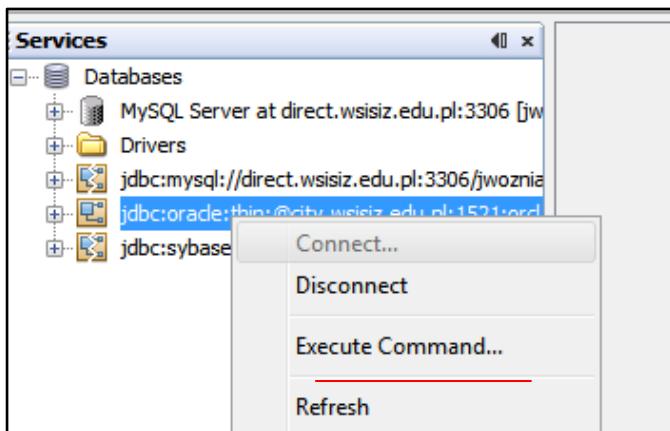


Po zatwierdzeniu definicji na liście połączeń widoczne będą obiekty bazodanowe wybranego schematu:



co oznacza, że połączenie zostało nawiązane.

3. Wybierając prawym przyciskiem myszy zdefiniowane połączenie uruchomić *Execute Command* i wykonać dowolne zdanie *select*.

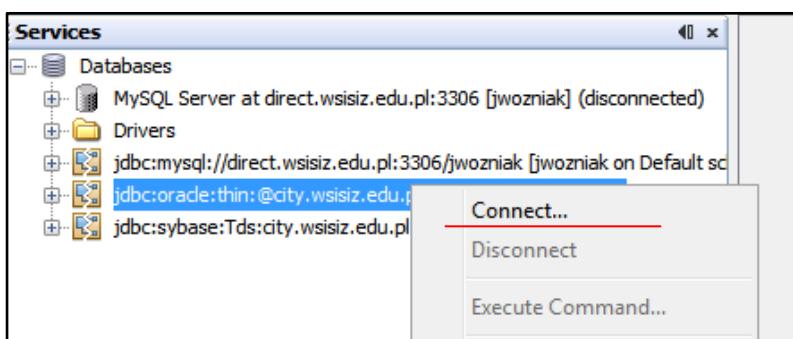


Szczególną uwagę zwrócić na zawartość tabel modelu BD4_RACHUNEK, gdyż na jego podstawie zaprezentowane zostanie tworzenie szablonu raportu typu ewidencyjnego².

4. Zakończyć pracę z *iReport Designer*.

III. Projektowanie raportu w formacie pdf

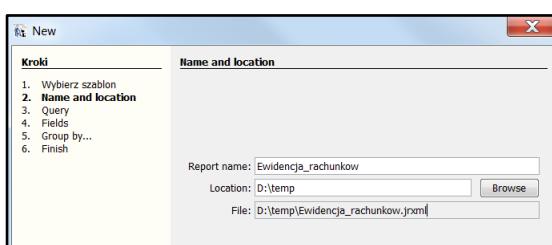
1. Uruchomić *iReport Designer* i w panelu Services uaktywnić prawym przyciskiem myszy połączenie z bazą danych na podstawie opracowanej wcześniej definicji jdbc :



Po rozwinięciu folderu *Tables* widoczne będą tabele schematu.

2. Z głównego menu należy wybrać funkcję *Plik/New...*, a następnie przycisk *Launch Report Wizard*. Przy pomocy uruchomionego kreatora wypełniać poszczególne formularze.

3. Na formularzu *Name and location* podać nazwę raportu oraz docelowy folder:



² Na liście tabel powinny znajdować się tabele z prefiksem BD4 utworzone wcześniej na podstawie materiału *Laboratorium BD7*. Dodatkowo w pliku *Laboratorium BD10.zip* zostały umieszczone odpowiednie skrypty implementujące ten model (*rachunek_create*, *rachunek_drop* i *rachunek_populate*).

4. Na formularzu *Query*, poprzez przycisk *New*, jako typ źródła danych wybrać *NetBeans Database JDBC connection*, a na kolejnym formularzu nazwać to połączenie, np.: {USER}_rachunki³ oraz z listy rozwijalnej wybrać odpowiednie. Przy pomocy przycisku *Test* sprawdzić poprawność połączenia. Przyciskiem *Save* zapisać tę definicję.

5. W polu *Query (SQL)* wpisać odpowiednie zdanie SQL, na podstawie którego będzie sporządzony raport, np.:

```
select nr_rachunku, data_sprzedazy, klient_nazwisko || ' ' || klient_imie klient,
       nazwa_produktu, ilosc_produktu, cena_produktu,
       ilosc_produktu * cena_produktu wartosc_rachunku
  from bd4_klient k
    join bd4_rachunek r on r.nr_klienta = k.nr_klienta
    join bd4_produkt p on p.nr_produktu = r.nr_produktu
   order by nr_rachunku      -- bez końcowego znaku ";"
```

(można wgrać z pliku *jasper_ewidencja.sql* z *Laboratorium BD10.zip*).

6. Na formularzu *Fields* wybrać wszystkie pola przyciskiem ">>", a formularz *Group by...* pozostawić bez zmian. Zakończyć pierwszy etap tworzenia raportu.

7. W panelu *Report Inspector* widoczna jest struktura opracowywanego dokumentu. Rozwinąć folder *Fields*. Powinny być widoczne wszystkie nazwy kolumn zawarte w zdaniu SQL, na podstawie których powstaje raport.

8. Z głównego menu rozwinąć funkcję *Preview* i wybrać *Internal Preview* jako format domyślny tworzonego raportu.

9. Na głównym pulpicie roboczym (w części środkowej ekranu) zwrócić uwagę na dwie zakładki: *Designer* oraz *Preview*. Pierwsza z nich to główny obszar roboczy tworzonego raportu, a druga to podgląd raportu w ustalonej formacie.

10. Metodą *Drag and Drop* należy przenieść odpowiednie pola z folderu *Fields* w panelu *Report Inspector* na pulpit roboczy *Designer* do sekcji *Detail 1*. Przykładowy wygląd wersji roboczej raportu może wyglądać tak:

Title						
Page Header						
Column Header						
NR_RACHUNKU	DATA_SPRZED	Klient	NAZWA_PRODUKTU	ILOSC_PRODUKTU	CENA_PRODUKTU	WARTOSC_RACUNKA
\$F	\$F	\$F{Klient}	\$F	\$F	\$F	\$F
Detail 1						

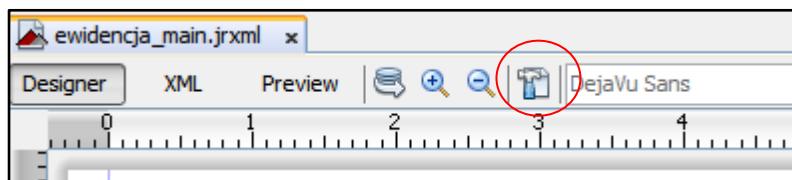
a po wybraniu zakładki *Preview* jako raport:

³ Zmienna USER oznacza nazwę zalogowanego użytkownika czyli pełna nazwa powinna wyglądać na przykład I7X4_01_rachunki. Jest to bardzo pomocne w przypadku, gdy na tej samej instalacji Jasper Reports będzie pracowało kilku użytkowników, na przykład w laboratoriach uczelnianych.

NR_RACHUNKU	DATA_SPRZED	Klient	NAZWA_PRODUKI	LOSC_PRODUK	CENA_PRODUK	WARTOSC_RAC
1	20.09.18 00:00	Abacki Adam	Canon 6D Body	4	4450	17800
2	05.10.18 00:00	Babacki Bogdan	Obiektyw EF	2	519.99	1039.98
3	20.09.18 00:00	Abacki Adam	Obiektyw EF 70-	3	3649	10947

Uwaga:

Wskazane jest, aby na pulpicie widoczny był obszar *iReport Output*. Jeśli tak nie jest należy z głównego menu wybrać *Okno / Report Output*. Po wykonaniu zmian w projekcie raportu można wykonać jego komplikację w celu upewnienia się, że projekt jest prawidłowy. Do tego celu służy przycisk:



11. Rozwinąć w głównym menu funkcję *Okno* i wybrać *Palette* (Ctrl+Shift+8).
12. Z *Palette* wybrać obiekt *Static Text* i położyć go w sekcji *Title* projektu raportu. Dwuklik na obiekcie umożliwia zmianę tekstu na "Ewidencja rachunków". Można to zrobić również w panelu *Właściwości*, który pokazuje zawsze właściwości aktualnie wybranego obiektu. Sformatować ten obiekt według uznania.
13. W podobny sposób sformatować nagłówki kolumn raportu, a pod etykietami wstawić linię przebiegającą przez całą długość wiersza. Nagłówki kolumn i kolumny z danymi tak rozmieścić, aby były widoczne wszystkie dane. Jeśli szerokość strony jest za mała można zmienić orientację z *Portrait* na *Landscape* (należy w Report Inspector wskazać nazwę szablonu (Ewidencja_rachunkow) i we właściwościach odnaleźć właściwość *Orientation*).

Ewidencja rachunków						
Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
1	20.09.18 00:00	Abacki Adam	Canon 6D Body	4	4450	17800
2	05.10.18 00:00	Babacki Bogdan	Obiektyw EF 50mm	2	519.99	1039.98
3	20.09.18 00:00	Abacki Adam	Obiektyw EF 70-200mm	3	3649	10947

14. W celu wstawienia obrazka (logo firmy) do sekcji *Title* należy z *Palette* wybrać obiekt *Image* i położyć go w lewej części sekcji, a następnie wybrać załączony, w *Laboratorium BD10.zip*, plik *my_logo.gif* (lub swój własny) i manualnie sformatować.

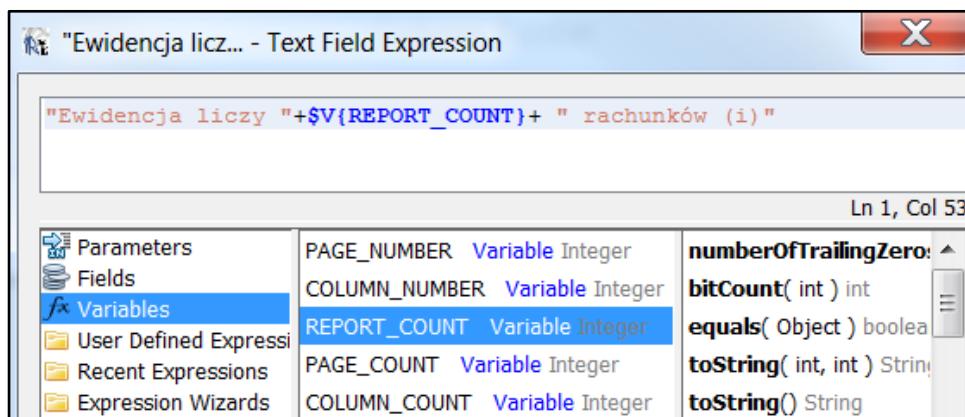
15. Z *Palette / Tools* wybrać *Current Date* i wstawić w prawym górnym rogu sekcji *Title* dobierając format daty według uznania.

16. W panelu *Report Inspector* rozwiniąć folder *Column Header*, odnaleźć obiekt *Line* stanowiący podkreślenie nagłówka raportu i metodą Kopij i Wklej utworzyć kopię w sekcji *Summary*. We właściwościach zmienić wartość w polach *Left* i *Top* na 0.

17. Z *Palette / Tools* wybrać *Page X of Y* i położyć z prawej strony sekcji *Page Footer*. We właściwościach tego obiektu znaleźć właściwość *Text Field Expression* i zmienić napis na język polski.

"Strona " + \$V{PAGE_NUMBER} + " z"

18. Z *Palette* wybrać *Text Field* i położyć go w sekcji *Summary*. We właściwościach rozwiniąć właściwość *Text Field Expression* i zdefiniować ją jako:



19. Dokonać końcowych modyfikacji raportu polegających na justowaniu nagłówków i danych, ustaleniu wielkości czcionki i typu czcionki czy wysokości każdej z sekcji oraz zmiany formatu wyświetlonej daty (może być przydatna właściwość *Pattern* dla danego pola).

20. Ostateczna postać raportu powinna wyglądać podobnie do przedstawionej poniżej:

Ewidencja rachunków						
06-12-2018						
 Ewidencja rachunków Demo						
<hr/>						
Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
1	20-09-2018	Abacki Adam	Canon 6D Body	4	4450	17800
2	05-10-2018	Babacki Bogdan	Obiektyw EF 50mm	2	519.99	1039.98
3	20-09-2018	Abacki Adam	Obiektyw EF 70-200mm	3	3649	10947
<hr/>						
Ewidencja liczby 3 rachunków (i)						

W menu głównym w sekcji *Preview* zmienić ustawienie na *PDF Preview* oraz powtórnie wykonać podgląd raportu (ikona *Run again* po prawej stronie zakładki *Preview*).

Można zauważyc, że w pliku pdf brak jest niektórych polskich czcionek:

Klient	Produkt	Ilo [szt]	Cena	Warto
Abacki Adam	Canon 6D Body	4	4450	17800
Babacki Bogdan	Obiektyw EF 50mm	2	519.99	1039.98

Zostanie to omówione w dalszej części materiałów.

IV. Definiowanie alternatywnych kolorów dla wierszy raportu

Dwoma najbardziej popularnymi metodami zwiększania czytelności danych w części *Detail* raportu są obramowywanie pól rapportu liniami o odpowiedniej grubości oraz naprzemienne kolorowanie wierszy przy pomocy zdefiniowanego koloru. Poniżej zostanie zaprezentowany ten drugi sposób.

- Wykorzystując opracowany wcześniej projekt rapportu *Ewidencja_rachunkow* zawierający siedem pól tekstowych, poprzez właściwości lub manualnie, dla sekcji *Detail 1* zwiększyć ponad dwukrotnie wysokość (*Band height*) sekcji (np. z 20 na 45), a następnie wprowadzić nowe pole tekstowe (*Text Field*) poniżej istniejących o długości takiej jak sumaryczna długość wszystkich istniejących pól.

Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
\$F	\$F	\$F{Klient}	\$F{NAZWA_PRODUKTU}\$F	\$F	\$F	\$F
\$F{field}						

- Ustawić właściwości tego pola, jak poniżej:

Left: taka sam jak wartość *Left* dla pola \$F{NR_RACHUNKU}

Height: taka sama jak wartość *Height* dla pól powyżej

Backcolor: wybrać kolor, np. [204,204,204] - szary

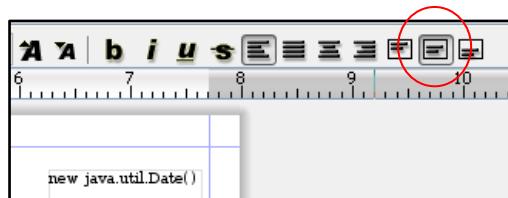
Opaque: ustawić

Print When Expression: \$V{REPORT_COUNT}.intValue() % 2 == 0

Text field Expressions: " "

- Przy pomocy prawego przycisku myszy na tym polu wybrać funkcję *Send To Back*.

- Zaznaczyć wszystkie pola z danymi (*Ctrl+myszka*) i wybrać dla nich wyśrodkowanie w poziomie przy pomocy przycisku pokazanego poniżej:



lub poprzez właściwości ustawić *Vertical Alignment* na *Middle*.

5. Odczytać wartość *Top* dla pól z danymi (powyżej) i taką samą wartość ustawić dla projektowanego pola. Zmienić wysokość sekcji *Details 1* tak, aby wiersz raportu wypełniał ją prawie całkowicie. Projekt powinien wyglądać podobnie do przedstawionego poniżej:

Ewidencja rachunków

Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
1	20-09-2018	Abacki Adam	Canon 6D Body	4	4450	17800
2	05-10-2018	Babacki Bogdan	Obiektyw EF 50mm	2	519.99	1039.98
3	20-09-2018	Abacki Adam	Obiektyw EF 70-200mm	3	3649	10947

Ewidencja liczy 3 rachunków (i)

6. Zmienić właściwość *Print When Expression* pola będącego barwnym paskiem (*Report Inspector* / sekcja *Detail 1* / pole tekstowe "") na: `$V{REPORT_COUNT}.intValue() % 2 == 1` i ponownie wygenerować raport.

Ewidencja rachunków

Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
1	20-09-2018	Abacki Adam	Canon 6D Body	4	4450	17800
2	05-10-2018	Babacki Bogdan	Obiektyw EF 50mm	2	519.99	1039.98
3	20-09-2018	Abacki Adam	Obiektyw EF 70-200mm	3	3649	10947

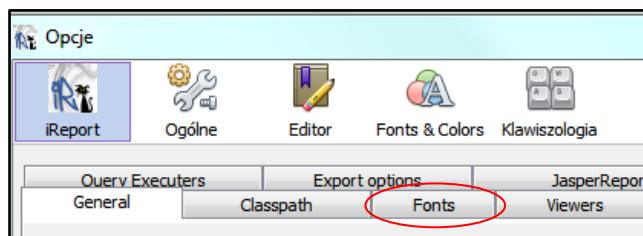
Ewidencja liczy 3 rachunków (i)

8. Zakończyć pracę z *iReport Designer*.

V. Zastosowanie dodatkowych bibliotek fontów przy projektowaniu i generowaniu raportów

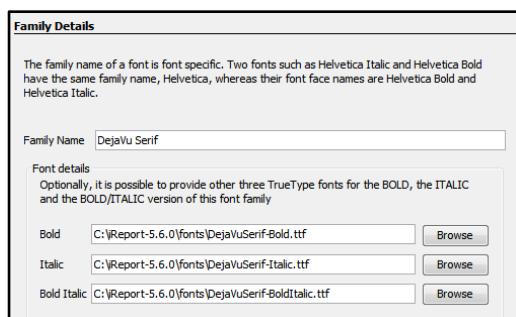
W *iReport Designer* możliwe jest używanie różnych rodzajów fontów. Zostanie to zaprezentowane na podstawie fontów z grupy DejaVu Serif.⁴

1. Ze strony <https://www.fontsquirrel.com/fonts/dejavu-serif> należy pobrać plik z fontami (*dejavu-serif.zip*) i rozpakować go w folderze roboczym.⁵
2. Uruchomić *iReport Designer* i przejść do funkcji *Narzędzia / Opcje* i zakładki *Fonts*.



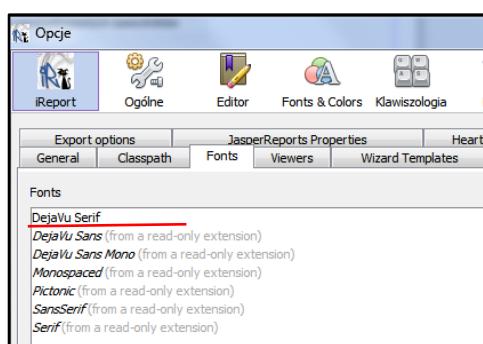
Przy pomocy przycisku *Install Font* uruchomić kreator instalacji fontów.

3. Na formularzu *Font selection* poprzez przycisk *Browse* wybrać plik *DejaVuSerif.ttf*. Na następnym formularzu *Family Details* uzupełnić pola dla fontów typu Bold, Italic i Bold Italic:



Pozostałe formularze pozostawić bez zmian. Zakończyć instalację.

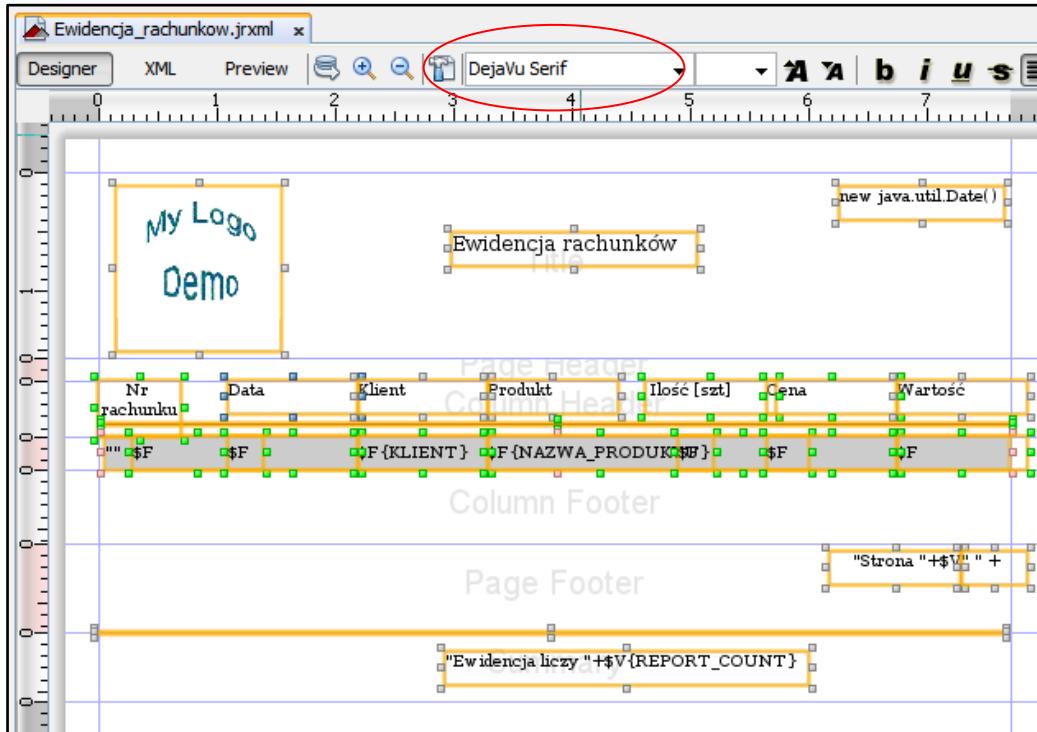
Zainstalowane fonty powinny być widoczne na liście *Fonts*:



⁴ Innym sposobem otrzymania polskich znaków w raporcie w formacie *pdf* jest ustawienie dla pól, w których te znaki występują lub mogą występować, właściwości *Pdf Encoding* na *CP 1250 (Central European)*. Wtedy instalowanie dodatkowych fontów tylko ze względu na polskie znaki jest niepotrzebne.

⁵ Fonty znajdują się również w *Laboratorium BD10.zip*.

4. Otworzyć projekt raportu *Ewidencja_rachunkow* i wybrać wszystkie jego elementy, a następnie po rozwinięciu listy fontów znaleźć zainstalowany font DejaVu Serif.



Dla wybranych elementów ustawić tę czcionkę.

5. Wygenerować raport. Można zauważyc zmianę typu czcionki oraz dodatkowo pojawię się polskich znaków w pliku pdf.

Nr rachunku	Data	Klient	Produkt	Ilość [szt]	Cena	Wartość
1	20-09-2018	Abacki Adam	Canon 6D Body	4	4450	17800
2	05-10-2018	Babacki	Obiektyw EF 50mm	2	519.99	1039.98
3	20-09-2018	Abacki Adam	Obiektyw EF 70-200mm	3	3649	10947

VI. Końcowe modyfikacje

- Na raporcie można zauważyc, że ilość danych w wierszu powoduje, że nie wszystkie kolumny wyświetlają się prawidłowo (na przykład kolumna z nazwiskiem i imieniem klienta). Można zmienić orientację raportu z *Portrait* na *Landscape* (należy w *Report Inspector* wskazać nazwę szablonu *Ewidencja_rachunkow* i we właściwościach odnaleźć właściwość *Orientation*).
- Dokonać odpowiednich modyfikacji polegających głównie na rozszerzeniu wszystkich pól w sekcji *Detail 1* wraz z ich nagłówkami oraz zmianie długości linii i barwnego paska różnicującego kolejne pozycje zestawienia.
- Przetestować dynamikę działania raportu poprzez wprowadzenie do bazy nowego rachunku oraz modyfikację istniejącego i wygenerowanie nowego raportu.

Na przykład:

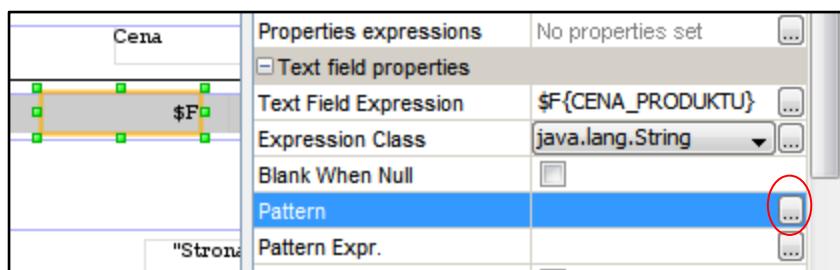
```
insert into bd4_rachunek
values (seq_rachunek.nextval, '2018/12/06', 5, 3055, 120, 'OTWARTE');

update bd4_rachunek
set data_sprzedazy = '2018/10/20'
where nr_rachunku = 3;

commit;
```

4. Pola zawierające cenę i wartość w każdym wierszu są typu tekstowego i dlatego są justowane standardowo do lewej strony. To, jak również format wyświetlania liczb, można zmienić.

Dla pola \${CENA_PRODUKTU} właściwość *Pattern* przed zmianą jest nieustawiona:



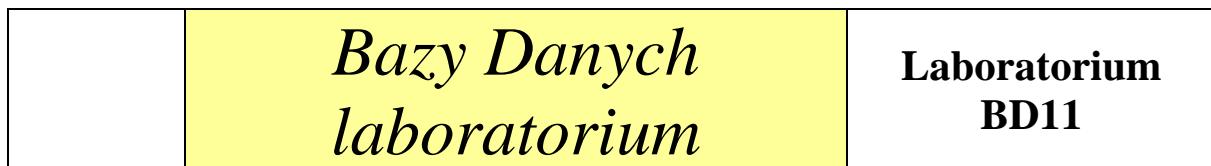
, wybierając przycisk "..." można ustawić *Custom Format* na #,###.00 oraz dodatkowo właściwość *Horizontal Alignment* na *Right* (justowanie do prawej). Tę drugą właściwość można również ustawić wybierając odpowiednią ikonę nad pulpitem roboczym.

Podobnie uczynić z polem \${WARTOSC_RACHUNKU}.

Ostateczny wygląd raportu się zmieni:

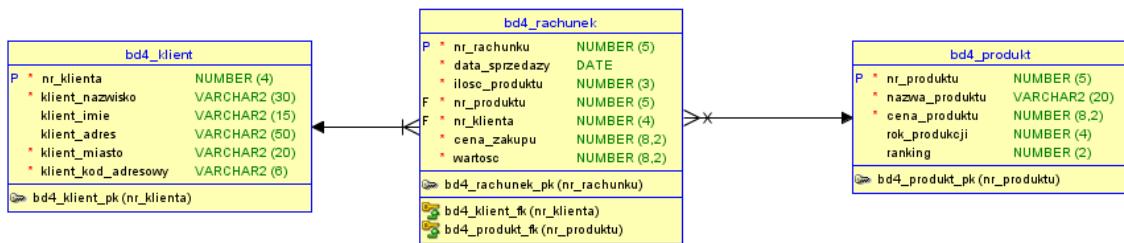
Cena	Wartość
4 450,00	17 800,00
519,99	1 039,98
3 649,00	10 947,00
519,99	2 599,95

5. Zakończyć pracę z *iReport Designer*.



Projektowanie programowego generatora danych dla modelu transakcyjnego

Na potrzeby tego laboratorium zostanie wykorzystany model BD4_RACHUNEK przedstawiony poniżej:



Należy go zaimplementować w swoim schemacie bazodanowym przy pomocy skryptów zawartych w *Laboratorium BD11.zip* lub zrealizować to samodzielnie¹.

Celem tego laboratorium jest opracowanie oprogramowania, które będzie generowało transakcje w sposób losowy symulując realizacje procesu sprzedaży.

I. Generowanie liczb pseudolosowych o rozkładzie równomiernym

Podobnie do innych języków programowania, język PL/SQL posiada oprogramowanie umożliwiające generowanie liczb pseudolosowych o rozkładzie równomiernym w zadanym przedziale wartości. Do tych celów został stworzony pakiet programowy *dbms_random* zawierający szereg funkcji służących generowaniu wartości numerycznych, dat i losowych ciągów znakowych.

W tym materiale omówione zostanie losowanie tylko wartości numerycznych.

W celu wylosowania liczby z przedziału [0..1] należy użyć funkcji:

```
select dbms_random.value from dual;
```

Przez wielokrotne wykonanie tego zapytania można otrzymać niepowtarzające się wartości liczbowe:

VALUE
0,06733144328319116994121925092753233857
0,38477513802834460982449269117469711753
0,09621358413832073950233266614131834018
0,00560247098923946115740914879052341913
....

z przedziału [0..1].

¹ Model ten różni się od opracowanego wcześniej modelu BD4. Chcąc z niego korzystać do budowy generatora danych należy poprzednią wersję usunąć skryptem *rachunek_drop*.

Chcąc zasymulować rzut kostką trzeba użyć innej formy tej funkcji:

```
select round ( dbms_random.value ( 1,6 ) ) "rzut kostką" from dual;
```

rzut kostką
4
2
3
5

....

W praktycznych zastosowaniach można spotkać taką konstrukcję:

```
select * from (
    select nr_produktu from bd4_produkt
        order by dbms_random.value)
where rownum = 1
```

NR_PRODUKTU
10
1
8
5

....

, w której podzapytanie losowo porządkuje zbiór numerów produktów, a zapytanie główne wybiera pierwszy numer produktu.

II. Koncepcja budowy generatora danych modelu fakturowania wielopozyycznego

Przedstawiony na diagramie model rachunku jednopozyycznego składa się z trzech tabel, z których dwie: BD4_KLIENT i BD4_PRODUKT mają charakter statyczny, to znaczy, że są one wypełnione danymi dotyczącymi klientów i produktów i zawartość ich nie musi się zmieniać.

Tabela BD4_RACHUNEK zawierająca transakcje kupna produktów przez klientów może być na początku pusta, a działanie generatora spowoduje, że programowo zostaną wypełnione danymi w dowolnej ilości.

Należy zatem jako czynność wstępną opracować skrypt zawierający zdania DDL tworzące wszystkie tabele modelu oraz inne niezbędne obiekty jak sekwencje i wyzwalacze bazodanowe. Model może być wzbogacony o inne tabele typu słownikowego, na przykład wymiarowanie klientów regionem zamieszkania, a produktów grupą asortymentową.

Drugim krokiem powinno być wprowadzenie do tabel BD4_KLIENT i BD4_PRODUKT (i ewentualnie słowników) danych demonstracyjnych w niewielkich ilościach (do 20 wierszy w tabeli).

Zadaniem generatora danych będzie możliwość utworzenia pełnej transakcji zakupu towaru przez klienta i zapisanie tej transakcji w tabeli BD4_RACHUNEK.

Poniżej zostanie zaprezentowana metoda tworzenia takiego oprogramowania metodą top_down (od ogółu do szczegółu).

Poziom 0:

Utworzenie pojedynczej transakcji będzie realizowane przez procedurę o specyfikacji:

```
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate);
```

Będzie ona odpowiedzialna za umieszczenie w tabeli BD4_RACHUNEK jednego wiersza stanowiącego kompletną transakcję.

Poziom 1:

Procedura pr_generuj_transakcje:

Powinna określić wszystkie niezbędne dane, aby możliwe było dokonanie wpisu do tabeli transakcyjnej.

Scenariusz może być następujący:

- NR_RACHUNKU - będzie wyznaczany sekwencją (na przykład seq_rachunek),
- DATA_SPRZEDAZY - parametr aktualny procedury,
- ILOSC_PRODUKTU – losowanie funkcją fn_daj_ilosc_produktu,
- NR_PRODUKTU – losowanie funkcją fn_daj_numer_produktu,
- NR_Klienta - losowanie funkcją fn_daj_numer_klienta,
- CENA_ZAKUPU – funkcja fn_daj_cene_produktu,

Uwagi:

1. Użycie sysdate jako daty transakcji jest prawidłowe, gdy generator będzie symułował działanie aplikacji czyli proces sprzedaży będzie rozłożony w czasie, na przykład przy wykorzystaniu automatów czasowych jakimi są joby. W przypadku użycia generatora do natychmiastowego wygenerowania określonej liczby transakcji należy odpowiednio ustawać wartość parametru aktualnego v_data_sprzedazy. Zostanie to omówione w dalszej części materiału.
2. Losowanie ilości zakupionego towaru musi być ograniczone do pewnego zakresu dopuszczalnych wartości (przedziału liczbowego) w zależności od asortymentu produktów.

Ogólny schemat tej procedury będzie wyglądał następująco:

```
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate);

begin
/*
wyznaczenie wszystkich niezbędnych wartości zgodnie z powyższym scenariuszem

v_nr_rachunku := seq_rachunek.nextval;
v_ilosc_produktu := fn_daj_ilosc_produktu (min_wartosc, max_wartosc);
v_nr_produktu := fn_daj_numer_produktu ();
v_cena_produktu := fn_daj_cene_produktu (v_nr_produktu);
v_wartosc := v_cena_produktu * v_ilosc_produktu;
v_nr_klienta := fn_daj_numer_klienta ();
```

oraz wpisanie ich do tabeli BD4_RACHUNEK zdaniem:

```
insert into bd4_rachunek values (v_nr_rachunku, v_data_sprzedazy
                                v_ilosc_produktu, v_nr_produktu, v_nr_klienta,
                                v_cena_produktu, v_wartosc);
commit;

*/
null;

end;
```

Poziom 2:

Procedura pr_generuj_transakcje zawiera w swoim kodzie funkcje odpowiedzialne za określenie odpowiednich wartości niezbędnych do utworzenia transakcji. Wszystkie one opierają będą swoje działanie o losowanie wartości zgodnie z rozkładem równomiernym czyli wylosowanie każdego klienta z tabeli BD4_Klient jest równoprawdopodobne. Analogicznie jest z produktami oraz ich ilością.

Funkcja fn_daj_numer_klienta:

Specyfikacja tej funkcji nie zawiera żadnych parametrów formalnych i zwraca wartość numeryczną:

```
function fn_daj_numer_klienta return numer;
```

Taka specyfikacja funkcji ma tę zaletę, że nie ma znaczenia, czy zbiór numerów klientów jest zbiorem ciągłym czy nie. Przy zastosowaniu techniki zaprezentowanej wcześniej kod tej funkcji zawiera zdanie SQL:

```
.....  
select * into v_nr_klienta from (  
    select nr_klienta from bd4_klient  
    order by dbms_random.value)  
where rownum = 1  
.....
```

Funkcja fn_daj_numer_produktu:

Specyfikacja tej funkcji jest analogiczna do funkcji *fn_daj_numer_klienta*:

```
function fn_daj_numer_produktu return numer;
```

Dodatkową zaletą może być uniezależnienie losowanej wartości od typu kolumny *nr_produktu*.

Jeśli kody produktów byłyby typu *varchar2*, np. A101, A102, B35,... to specyfikację tej funkcji należało by zmienić na:

```
function fn_daj_numer_produktu return varchar2;
```

, ale algorytm pozostał by ten sam.

Funkcja fn_daj_ilosc_produktu:

Ta funkcja musi mieć zdefiniowane parametry formalne, którymi jest wartość minimalna i maksymalna ilości produktu:

```
function fn_daj_ilosc_produktu ( min_ilosc number default 1,  
                                max_ilosc number default 10 ) return number;
```

W tej specyfikacji standardowo ustawiona jest minimalna i maksymalna wartość, co oznacza, że wywołanie jej może odbywać się na kilka sposobów:

```
v_ilosc_produktu := fn_daj_ilosc_produktu ();  
v_ilosc_produktu := fn_daj_ilosc_produktu (5, 30);  
v_ilosc_produktu := fn_daj_ilosc_produktu (max_ilosc => 15);
```

Funkcja fn_daj_cene_produktu:

Ta funkcja ma pobierać cenę wylosowanego wcześniej produktu z tabeli BD4_PRODUKT z ewentualną możliwością symulacji negocjowania rabatu:

```
function fn_daj_cene_produktu ( v_numer_produktu number ) return number;
```

Podsumowanie:

W wyniku zastosowania takiej metody dekompozycji oprogramowania można wyspecyfikować wszystkie obiekty programowe niezbędne do zbudowania generatora danych.
Są to:

```
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate);  
  
function fn_daj_numer_klienta return numer;  
  
function fn_daj_numer_produktu return numer;  
  
function fn_daj_ilosc_produktu ( min_ilosc number default 1,  
                                 max_ilosc number default 10 ) return number;  
  
function fn_daj_cene_produktu ( v_numer_produktu number ) return number;
```

III. Zastosowanie pakietów PL/SQL do budowy generatora danych**Informacje wstępne:**

Pakietы PL/SQL grupują logicznie powiązane składniki, takie jak, zmienne, struktury danych, wyjątki oraz procedury i funkcje. Składają się z dwóch części: specyfikacji i implementacji.

Umożliwiają serwerowi Oracle jednocześnie wczytywanie wielu obiektów do pamięci. Pakiet nie może być wywoływany, zagnieżdżany ani nie może mieć parametrów. Po napisaniu i skompilowaniu pakietu jego zawartość może być współużytkowana przez wiele aplikacji czy sesji.

W momencie pierwszego odwołania się do pakietu jest on w całości ładowany do pamięci i następne próby używania konstrukcji zawartych w pakiecie nie wymagają już operacji dyskowych wejścia-wyjścia.

Budowa pakietu:

Specyfikacja pakietu definiowana jest jak poniżej:

```
create or replace package pkg_generator_danych is  
  
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate);  
  
function fn_daj_numer_klienta return number;  
  
function fn_daj_numer_produktu return number;  
  
function fn_daj_ilosc_produktu ( min_ilosc number default 1,  
                                 max_ilosc number default 10 ) return number;  
  
function fn_daj_cene_produktu ( v_numer_produktu number ) return number;  
  
end pkg_generator_danych;
```

Zgłaszać są specyfikacje funkcji i procedur. Dodatkowo można deklarować zmienne, własne typy danych, na przykład rekordowe lub tablicowe.

Tylko obiekty zdefiniowane w specyfikacji pakietu będą widoczne przez inne aplikacje czy też sesje.

Sposób odwoływania się do obiektu pakietowego wygląda tak:

```
-- funkcja pakietowa
v_nr_klienta := pkg_generator_danych.fn_daj_numer_klienta;
```

lub

```
-- procedura pakietowa
begin
    pkg_generator_danych.pr_generuj_transakcje ( '2020/09/18' );
end;
```

Implementacja pakietu definiowana jest jak poniżej:

```
create or replace package body pkg_generator_danych is

procedure pr_generuj_transakcje (v_data_sprzedazy default sysdate)
begin
    null;
    /* implementacja procedury */
end pr_generuj_transakcje;

function fn_daj_numer_klienta return number;
begin
    return null;
    /* implementacja funkcji */
end fn_daj Numer_klienta ;
.....      -- kod wszystkich procedur i funkcji zdefiniowanych w specyfikacji
              -- pakietu
end pkg_generator_danych;
```

Usuwanie pakietu:

Aby usunąć specyfikację i implementację pakietu należy użyć zdania:

```
drop package pkg_generator_danych;
```

Aby usunąć tylko implementację pakietu należy użyć zdania:

```
drop package body pkg_generator_danych;
```

Uwagi końcowe:

1. Nie można usunąć specyfikacji pakietu zostawiając jego implementację.
2. Procedury i funkcje w pakietach mogą być przeciążane, co oznacza, że można projektować obiekty o tych samych nazwach, lecz o argumentach formalnych różniących się istotnie między sobą.

Na przykład:

```
-- procedura generująca jedną transakcję poprzez generator
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate);
```

```
-- procedura generująca wiele transakcji z jedną datą poprzez generator
procedure pr_generuj_transakcje (v_data_sprzedazy date default sysdate,
v_ilie number);
```

```
-- procedura generująca wiele transakcji w zadanym przedziale czasu poprzez generator
procedure pr_generuj_transakcje (pocz_data date, konc_data date);
```

Przykłady zastosowania i sposoby wywoływania tych procedur:

- wygenerowanie jednej transakcji z określona datą:

```
begin
  pkg_generator_danych.pr_generuj_transakcje ('2020/09/18');
end;
```

- wygenerowanie kilku transakcji z tą samą datą:

```
begin
  pkg_generator_danych.pr_generuj_transakcje
    (v_data_sprzedazy => '2020/09/18',
     v_ilie => 10);
end;
```

- wygenerowanie wielu transakcji w określonym przedziale czasowym:

```
begin
  pkg_generator_danych.pr_generuj_transakcje
    (pocz_data => '2020/01/01',
     konc_data => sysdate);
end;
```

W tym przypadku podstawą algorytmu generowania powinno być symulowanie upływu czasu czyli wyznaczanie w pierwszej kolejności daty i dla niej generowanie określonej liczby transakcji:

```
cur_date := pocz_data;
while cur_date <= konc_data
  loop
    pkg_generator_danych.pr_generuj_transakcje
      (v_data_sprzedazy => cur_date,
       v_ilie => 10);
    cur_date := cur_date + 1;
  end loop;
```

Warto zwrócić uwagę, że w takim przypadku będziemy mieć do czynienia z wywoływaniem w procedurze *pr_generuj_transakcje* przeciążonej procedury.

Dodatkowo można utworzyć pakietową funkcję *fn_daj_liczbe_transakcji*, która będzie losowo określała liczbę transakcji dla każdego dnia i wtedy w powyższym algorytmie drugi argument procedury może wyglądać tak:

```
.....
v_ilie => fn_daj_liczbe_transakcji (min_laczba => 1,
                                         max_laczba => 10) ;
```

```
.....
```

- Można opracować automat czasowy (*job*), który z określonym interwałem czasowym (na przykład co jedną godzinę) będzie uruchamiał procedurę *pr_generuj_transakcje* z bieżącą datą. Do tego celu służy pakiet programowy *dbms_scheduler*, przy pomocy którego można definiować różne warianty realizacji zadań.

Zadania do samodzielnego wykonania:

1. Opracować i przetestować kompletny pakiet generatora danych omówiony w tym materiale.

Scenariusz postępowania:

- usunięcie z tabeli BD4_RACHUNEK wszystkich wierszy,
- skasowanie sekwencji seq_rachunek,
- założenie sekwencji seq_rachunek,
- uruchomienie generatora:

```
begin
    pkg_generator_danych.pr_generuj_transakcje
        (pocz_data => '2020/01/01', konc_data => sysdate);
end;
```

- (opcjonalnie) opracowanie automatu czasowego (job), który z określonym interwałem będzie uruchamiał procedurę:

```
begin
    pkg_generator_danych.pr_generuj_transakcje (sysdate);
end;
```

2. Poddać analizie prawidłowość losowania liczb pseudolosowych przy pomocy pakietu dbms_random. W tym celu należy założyć tabelę testową, w której sumowane będą wylosowane wartości.

Scenariusz tego eksperymentu w oparciu o rzut kostką:

- założenie tabeli zawierającej trzy kolumny: wylosowana wartość, ilość wystąpień, procentowy rozkład,
- wpisanie do niej sześciu wierszy (x, 0, 0), gdzie x - wartość [1..6],
- napisanie bloku PL/SQL lub procedury, wykonującej określoną liczbę losowań (na przykład 6000) i aktualizującą po każdym losowaniu kolumnę ilość wystąpień dla wylosowanej wartości,
- zaktualizowanie kolumny procentowy rozkład.

3. Na podstawie zasad budowy zadań zawartych w *Laboratorium BD9* oraz skryptu *job_generowanie_transakcji.sql* (*Laboratorium BD11.zip*) opracować zadanie (job) generujące pojedynczą transakcję z określonym interwałem czasowym zawartym w przedziale [3 ..6] godzin.

Założenia projektu zaliczeniowego

Cel:

Kompletne opracowanie projektu zaliczeniowego, na który składa się implementacja w środowisku Oracle modelu relacyjnej bazy danych, oprogramowanie realizujące proces automatycznego generowania danych wejściowych oraz prezentujące wyniki w postaci analitycznej.

Harmonogram projektu:

1. Opracowanie modelu bazy danych, w oparciu o który realizowane będą dalsze części projektu.
2. Implementacja modelu na udostępnionym serwerze Oracle oraz opracowanie programowego symulatora generującego dane demonstracyjne.
3. Opracowanie skryptów wdrożeniowych służących do implementacji i zarządzania projektem.
4. Opracowanie dokumentacji projektu.

Termin zakończenia projektu:

Termin zasadniczy - trzy dni przed ostatnimi zajęciami.

Szczegółowe założenia projektowe:

1. Opracować model bazy danych dowolnego fragmentu rzeczywistości zawierający elementy transakcyjne (sprzedaż, wypożyczenie, naprawa itp.). Dodatkowo strony transakcji podlegają wymiarowaniu poprzez tabele słownikowe (grupy produktowe, województwa, marki samochodów itp.). Do realizacji tego punktu należy wykorzystać narzędzie Oracle Data Modeler. Model powinien być zaimplementowany na serwerze Oracle 12c (city.wsisz.edu.pl). Model logiczny powinien zawierać do 10 encji będących w związkach M:N (obowiązkowo) i 1:N, w którym występują atrybuty służące do liczenia (koszty, zyski, punkty itp.).
2. Implementacja powinna zawierać dwie metody generowania kluczy głównych tabel, tzn. sekwencje i wyzwalacze. Dodatkowo wyzwalacze powinny zawierać elementy zautomatyzowanych obliczeń zgodnie ze swoim przeznaczeniem, zapewniając spójność bazy danych.
3. Do wszystkich tabel należy wprowadzić sensowne dane.
4. Opracować w postaci minimum czterech perspektyw wyniki zbiorcze obrazujące efekty działania zaprojektowanego oprogramowania na podstawie wprowadzonych danych. Perspektywy powinny pokazywać typowe wskaźniki biznesowe, na przykład zyski z prowadzonej działalności, wielkość sprzedaży poszczególnych produktów w funkcji czasu (lata, kwartały czy miesiące). Perspektywy powinny być zbudowane w sensowny sposób w oparciu o co najmniej trzy tabele. Można wykorzystać również zmienne wiązania symulując działanie w aplikacji. Na podstawie wybranych dwóch perspektyw należy utworzyć raporty w środowisku JasperReports (typu zbiorczego zestawienia transakcji oraz analitycznego odpowiednio sparametryzowanych).
5. Opracować sparametryzowane trzy funkcje służące celom obliczeniowym (w tym jedną logiczną) i trzy procedury zapewniające wprowadzanie, aktualizację i kasowanie danych w bazie. Oprogramowanie to powinno zawierać walidację danych i generować stosowne komunikaty końcowe.
6. Opracować skrypty wdrożeniowe umożliwiające instalację i deinstalację projektu na dowolnym koncie. Skrypt instalujący powinien zawierać zdania SQL tworzące obiekty bazodanowe (tabele, sekwencje, perspektywy, wyzwalacze, procedury i funkcje) oraz zdania wprowadzające dane.

Dopuszcza się możliwość istnienia kilku skryptów, na przykład tworzenie obiektów i wprowadzanie danych do tabel. Skrypt deinstalujący powinien usunąć ze schematu cały projekt.

7. W postaci zdań zademonstrować użycie perspektyw, zdań podrzędnych nieskorelowanych umieszczonych we frazach from, where i having i zawierających funkcje agregujące i limitowanie wierszy (rownum). Można zastosować zmienne wiązania.
8. Opracować dokumentację projektową zawierającą:
 - 8.1. Analizę biznesową projektowanej rzeczywistości,
 - 8.2. Model logiczny i relacyjny bazy danych,
 - 8.3. Oprogramowanie tworzące bazę danych,
 - 8.4. Skrypty wdrożeniowe instalujące i deinstalujące zrealizowany projekt,
 - 8.5. Instrukcję instalacji projektu i sprawdzenia jego poprawności,
 - 8.6. Wyniki działania perspektyw w postaci raportów pdf.

Dodatkowe założenia (nieobowiązkowe):

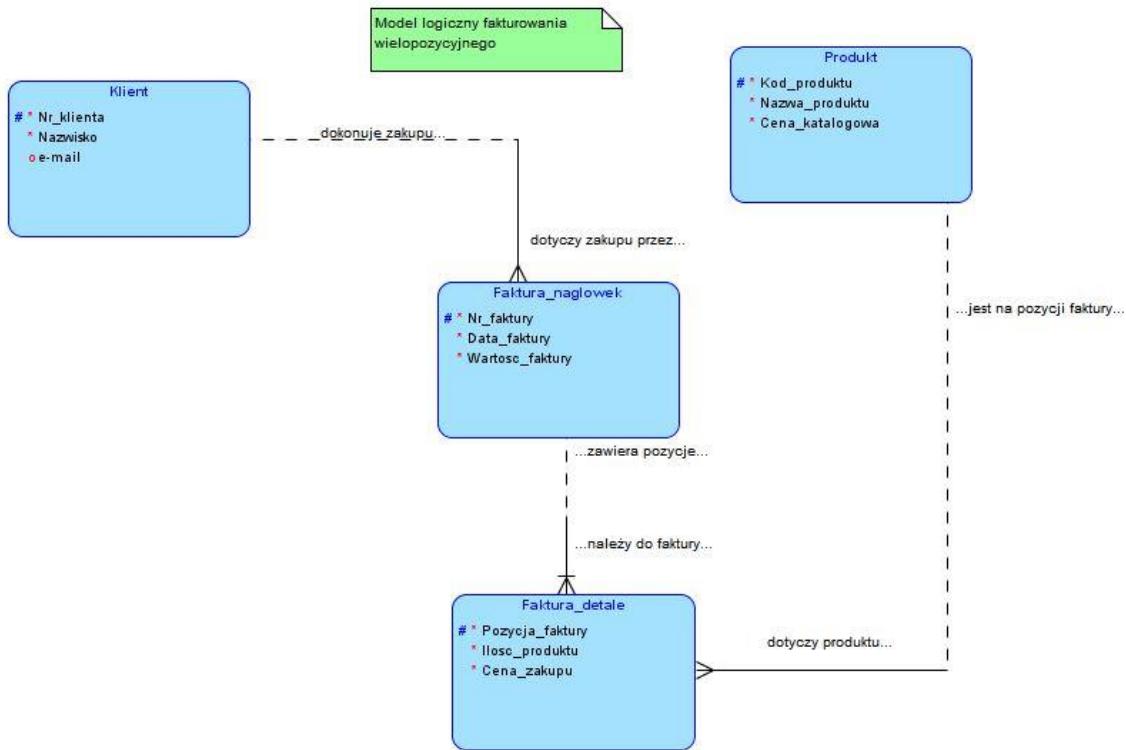
9. Opracować programowy generator danych przy pomocy języka PL/SQL. Generator powinien w sposób automatyczny lub manualny generować transakcje na podstawie wprowadzonych danych stałych (strony transakcji i dane słownikowe). Zbudowany powinien być ze sparametryzowanych procedur i funkcji PL/SQL (ewentualnie zawierających zmienne wiązania). Do realizacji tego punktu w sposób automatyczny można wykorzystać obiekty programowe jobs.
10. Zastępco, zamiast tworzenia raportów przy pomocy Jasper Reports i programowego generatora danych, można wykonać fragment aplikacji w Oracle Application Express. Wymaga to wcześniejszego uzgodnienia w celu przygotowania środowiska programowego oraz omówienia założeń na tworzoną aplikację.

Materiały pomocnicze:

Typowa faktura zakupowa składa się z elementów przedstawionych na poniższym rysunku:

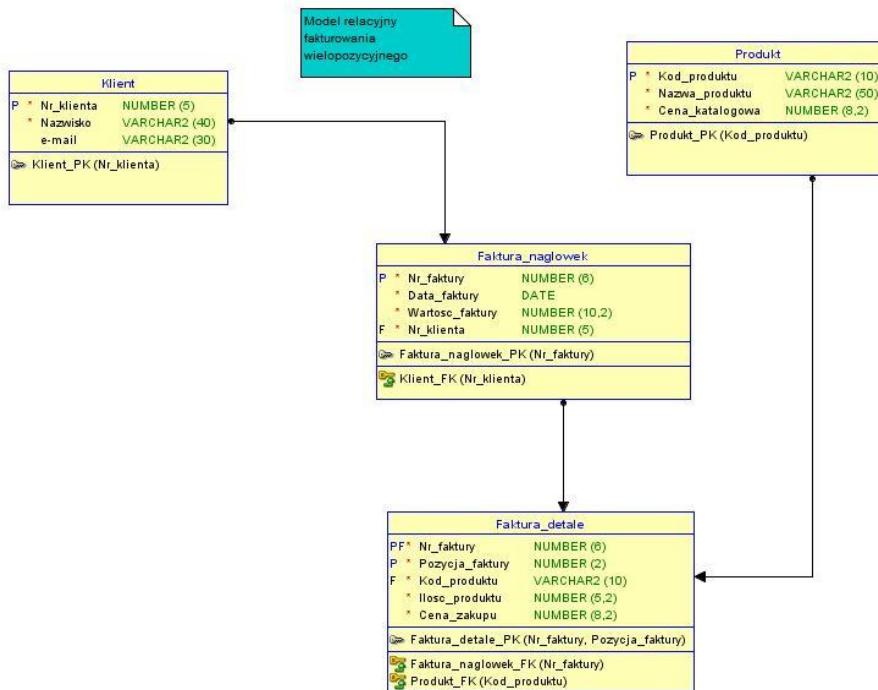
Faktura Nr 234/2015							Data wystawienia: 2015-11-30
Firma sprzedająca:							Odbiorca:
ABC Soft 01-345 Warszawa ul. Bliska 15/40							Adam Abacki 01-897 Warszawa ul. Kwitnąca 34/4
Wartość faktury: 2373,90 zł							
Termin płatności: 2015-12-29							
Poz.	Nazwa produktu	Cena	Ilość	%VAT	VAT	Brutto	
1	Monitor	750,00 zł	1	23%	172,50 zł	922,50 zł	
2	Dysk	340,00 zł	2	23%	156,40 zł	836,40 zł	
3	Drukarka	250,00 zł	2	23%	115,00 zł	615,00 zł	
Razem:							2 373,90 zł

Typowy model fakturowania wielopozycyjnego dotyczący sprzedaży towarów zarejestrowanym klientom można przedstawić poniższym modelem logicznym:

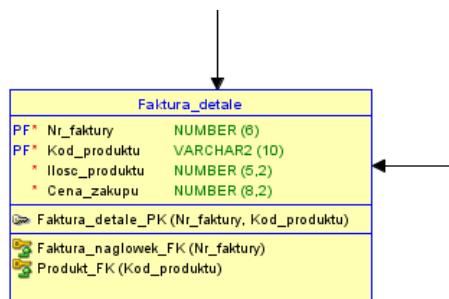


Model ten zawiera dwie encje będące stronami transakcji (Klient i Produkt) oraz dwie (pozostałe) przedstawiające model fakturowania wielopozyycjnej. Wszystkie związki zaznaczone na powyższym diagramie są typu 1:N. Klient może mieć wystawionych wiele faktur, produkt może występować na wielu pozycjach faktur, a jedna faktura może mieć wiele pozycji.

Odpowiadający powyższemu relacyjny model wyglądać będzie tak:



Drugim wariantem tego modelu może być brak pozycji faktury i wtedy kluczem głównym tabeli Faktura_detalie będzie numer faktury i kod produktu.



Pierwsze rozwiązanie ma tę zaletę, że kolejność pozycji na fakturze jest określana przez operatora, na przykład według ważności pozycji (przy kupnie samochodu najpierw samochód, a potem płyn do spryskiwacza). Wadą jest możliwość zdublowania danej pozycji na fakturze. Drugie rozwiązanie eliminuje możliwość wystąpienia powtarzających się pozycji, ale kolejność na fakturze może być przypadkowa i zależna od kodu produktu.

Założenia alternatywne (dla studentów studiów zaocznych):

1. Założenia na system informatyczny (pod kątem modelu bazy danych):

System informatyczny do obsługi ośrodka szkolenia kierowców jest oparty na bazie danych, w której przechowywane są dane kursantów – m.in. dane personalne, wybrana kategoria prawa jazdy, preferencje dotyczące lokalizacji zajęć, listy dostarczonych dokumentów oraz dokonanych i zaległych opłat; dane instruktorów – m.in. dane personalne, preferowana lokalizacja zajęć, informacja na temat możliwości prowadzenia zajęć praktycznych i wykładów teoretycznych dla poszczególnych kategorii prawa jazdy oraz dane samochodów – m.in. dane techniczne, informacja o możliwości prowadzenia zajęć dla poszczególnych kategorii prawa jazdy, informacje o naprawach oraz planowanych przeglądach. Należy stworzyć możliwość monitorowania o zbliżających się terminach obowiązkowych przeglądów oraz o konieczności odbioru naprawionych samochodów.

System powinien przechowywać również informacje o planowanych terminach kursów oraz udostępniać informacje o wolnych terminach szkoleń. Zapisy na kurs mogą odbywać się sposobem tradycyjnym lub mailowo. Jeśli kursant poda swój adres email, każda rejestracja będzie automatycznie potwierdzana drogą mailową.

W systemie mają być przechowywane również dodatkowe informacje na temat przeprowadzonych kursów, na przykład przechowywane będą sugestie uczestników kursów w postaci ankiet na temat ich przebiegu oraz możliwych usprawnień.

Po przeprowadzeniu każdego egzaminu wewnętrznego zapisywany będzie jego wynik oraz w przypadku wyniku negatywnego – powód niezaliczenia. Zakłada się, że ośrodek będzie otrzymywać z zewnętrznego źródła wyniki egzaminów państwowych swoich kursantów. Wyniki te będą wprowadzane i przechowywane w systemie. Na podstawie tych danych prowadzone będą statystyki najczęstszych problemów, na podstawie których będzie można dostosować optymalny plan kursów.

2. Przy pomocy Oracle Data Modeler opracować model logiczny i relacyjny projektowanego systemu.
3. W modelu opracować perspektywy dedykowane osobom funkcyjnym ośrodka, takim jak *rejestratorka kursantów, nadzorca samochodów i kierownik ośrodka, kursant*.
4. Wykonać dokumentację projektu, na którą składać się będą założenia na projektowany system, przyjęte rozwiązania w postaci modelu logicznego i relacyjnego wraz ze stosownym opisem oraz wygenerowany skrypt realizujący implementację projektu na jednym z możliwych serwerów bazodanowych.