Wprowadzenie do baz danych

Notatki z zajęć

Arkadiusz Ostrzyżek

Spis treści

Podstawy tabel

Tworzenie tabeli

Tabele tworzone są przy użyciu **create table TABELA**. Każda tabela powinna mieć klucz główny, który będzie stanowić numer identyfikacyjny danego elementu w 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 tworzeniu tabel możemy przydzielić kolumnie restrykcje ograniczającą dane jakie może przechowywać.

```
create table nazwa_tabeli (kol_a typ_danych Primary Key,
                           kol_b typ_danych not null
);
```

Klucz złożony może być tylko zdefiniowany na poziomie tabeli! Zamiast zostać utworzonym na początku, tworzy się go podobnie do Foreign Key.

```
Primary Key(kol_a, kol_b)
```

Modyfikacje tabeli

Usuwanie kolumny z tabeli

Wszystkie modyfikacje wymagają użycia **alter table NAZWA_TABELI**, po czym uzupełniane.

cel	sposób
dodawanie kolumny	add KOLUMNA TYP_ZMIENNEJ;
usuwanie kolumny	drop column KOLUMNA;
typu danych kolumny	modify KOLUMNA TYP_ZMIENNEJ;
zmiana nazwy kolumny	rename column KOLUMNA_1 to KOLUMNA_2;
zdefiniowanie referencji	add constraint FK_1 foreign key (PK_2) references TABLE_2 (PK_2);
usunięcie referencji	drop constraint FK;

Wyjątkami od tego są zmiany nazwy tabeli oraz usunięcie tabeli z bazy danych. **rename**

TABELA_1 TO TABELA_2; oraz **drop table TABELA**.

Manipulacja danych

Insert

W celu dodawania danych używamy **insert TABLE**. Domyślnie zakłada ono, iż wprowadzamy dane zgodnie z kolejnością występowania w tabeli. Aby ominąć podawania danej, można użyć null.

```
insert into TABELA
values
    (VAR1, VAR2, VAR3);
```

Wartości można wprowadzić także w innej kolejności, po zdefiniowaniu ich występowania. Tą technikę można też użyć w celu uzupełnienia tylko części danych.

```
insert into TABELA
    (KOLUMNA_VAR2, KOLUMNA_VAR3, KOLUMNA_VAR1)
values
    (VAR2, VAR3, VAR1);
```

Update

W celu modyfikowania danych używamy **update TABLE**. Możemy nim zmieniać dane w poszczególnych wierszach używając WHERE, będącego odpowiednikiem if-statement.

```
update TABLE
    set KOLUMNA_2 = 9,
        KOLUMNA_3 = 5
where KOLUMNA_1 = 3;
```

update można też użyć bez **where** w celu zmienienia wszystkich wierszy w danej kolumnie.

```
update TABLICA
    set KOLUMNA_2 = 9;
```

Delete

delete możemy używać w analogiczny sposób co **update**.

```
delete TABELA
    where KOLUMNA_2 = 5;
```

Użycie **delete TABELA** spowoduje natomiast usunięcie całej tabeli.

Commit / rollback

commit zatwierdza wszelkie zmiany i ustanowi nowy punkt wyjściowy. **rollback** cofa nas do ostatniego momentu użycia commit, bądź jeśli taki nie istnieje, do początku sesji.

WAŻNE: opcje **create, alter, drop rename** są równoznaczne z użyciem **commit**.

Pozyskiwanie informacji

Select

Zdanie **select** używane jest do wybierania konkretnych table z tabeli. Dla przykładu jesteśmy w stanie wyświetlić całą zawartość tabeli.

```
1 select * from TABELA;
```

w celu zliczenia wszystkich kolumn można zastosować zdania **count()**.

```
1 select count(*) from TABELA
```

Jest ono jednak używane w celu uzyskania bardziej konkretnych informacji. Jesteśmy w stanie wybrać kolumny jakie chcemy rozpatrzeć w ramach naszego zapytania.

```
1 select KOLUMNA_1, KOLUMNA_2 from TABELA;
```

Może to prowadzić jednak do stworzenia się powtarzających się danych, jeśli nasze kolumny nie zawierają klucza głównego, a więc w celu zachowania unikatowości wyników możemy użyć **discinct**. Odnosi się ono do układu wszystkich kolumn.

```
1 select distinct KOLUMNA_2, KOLUMNA_3 from TABELA;
```

Tak samo jak w przypadku **update**, jesteśmy w stanie dodać **where** w celu zawężenia naszego wyszukania, opisanego dokładniej poniżej

```
1 select distinct KOLUMNA_2, KOLUMNA 3
2     from TABELA
3     where KOLUMNA_2 = 9;
```

Nazwy kolumn można zmienić używając **as**.

```
1 select distinct KOLUMNA_2 as "KOLUMNA"
2     from TABELA;
```

Używając **select** jesteśmy też w stanie złączyć dwie kolumny przy wyświetlaniu.

```
1 select nazwisko || ' ' || imie as Imie
2     from TABELA;
```

Jesteśmy też w stanie połączyć tekst i cyfrę używając **cast**.

```
1 select 'żPoniej ąs ' || cast(count(*) as VARCHAR2(50)) || '
    wyniki'
2 from TABELA;
```

Where

Zadania logiczne zawarte w where mogą być bardziej złożone, gdyż dozwolone jest używanie **and**, **not**, **or** oraz <, >, <= etc.

```
1
2 select * from TABLE
3       where not
4       (
5       (KOLUMNA_3 > KOLUMNA_2 or KOLUMNA_1 > KOLUMNA_2)
6       and
7       (KOLUMNA_3 = 5)
8       );
```

operator	znaczenie	przykład
between	wartości z przedziału	where KOLUMNA between 3 and 5
in	zawarta w liście	where KOLUMNA in ('A', 'B', 'C')
like	regex	where KOLUMNA like 'T_o%'

Warto zaznaczyć też różnicę pomiędzy **count(*)** oraz **count(KOLUMNA_1)**. Obie opcje mogą zostać użyte do zliczania, jednak opcja pierwsza będzie zliczać także wyniki z wartością równą NULL, a druga już nie.

Zmienne wiązania

Możemy pozwolić użytkownikowi na wpisywanie danych używając symbolu & przed zmienną. Wywołanie takiego zapytania spowoduje pojawienie się okna, czekającego na input użytkownika.

```
1 select * from TABELA
2 where KOLUMNA_1 = &numer;
```

Zarządzanie NULL

Warto wiedzieć, że: 1. Null nie jest wartością. 2. Null nie jest tożsamy z 0. 3. Null jest nie większy, nie mniejszy i nie równy dowolnej wartości. 4. Null nie jest równy null. 5. Operacje matematyczne z null, są równe null. 6. **select null or true** jest równe true. 7. **select null or false** jest równe null. 8. **select null and true** jest równe null. 9. **select null and false** jest równe false.

Aby wybrać null zdaniem **where** należy użyć **is**.

```
1 delete TABELA
2       where KOLUMNA is null;
```

Sortowanie

Domyślnie baza danych nie jest uporządkowana w żaden sposób. W celu uporządkowania używa się zdania **order by**.

```
select *
    from TABLICA
    order by KOLUMNA;
```

Możliwe jest też posortowanie w odwrotnej kolejności poprzez dodanie **desc**.

```
select *
    from TABLICA
    order by KOLUMNA desc;
```

W celu umieszczenia wszelkich null na początku zbioru można użyć **nulls first**.

```
select *
    from TABLICA
    order by KOLUMNA nulls first;
```

Nie można sortować po wartości, której nie wybieramy, przy użyciu distinct. W tym celu trzeba dodać **max**.

```
select KOLUMNA_1
    from TABLICA
    group by KOLUMNA_1
    order by max(KOLUMNA_2);
```

W tym przypadku **group by** zadziała tak samo, co **distinct**.

Łączenie tabel

Wszystkie funkcje typu 'join' używane są w celu łączenia tabel. Pozwala nam to na tworzenie zestawień, które będą zawierały potrzebne dla nas w danym momencie informacje z różnych tabel. Wszystkie rodzaje table można uzyskać używając **using** lub **join on**. W kodzie należy zastąpić dane w <> naszym typem złączenia.

W przypadku nie powtarzających się nazw kolumn, nie trzeba precyzować wyboru tabeli albo używać **using**, w przeciwieństwie do naszego przypadku.

```
select KOLUMNA_1, KOLUMNA_2, KOLUMNA_3
from TABLICA_1
<TYP ŁĄZCZENIA> join tablica_2
using (KOLUMNA_1);
```

```
select t1.KOLUMNA_1, t1.KOLUMNA_2, t2.KOLUMNA_3
from TABLICA_1 t1
<TYP ŁĄZCZENIA> join TABLICA_2 t2 on t1.KOLUMNA_1 = t2.
    KOLUMNA_1;
```

Jedynym wyjątkiem będzie full join, który wymaga użycia **coalesce**.

```
select COALESCE(t1.KOLUMNA_1, t2.KOLUMNA_1) as KOLUMNA_1, t1.
    KOLUMNA_2, t2.KOLUMNA_3
from TABLICA_1 t1
full join TABLICA_2 t2 on t1.KOLUMNA_1 = t2.KOLUMNA_1;
```

Warto zwrócić tutaj uwagę na użycie aliasów. Należy pamiętać, iż niedopuszczalne jest w takim przypadku używanie w jednym zdaniu select zarówno pełnej nazwy tabeli, jak i jej aliasu.

Przykładowe tabele używane do demonstracji:

KOLUMNA_1	KOLUMNA_2
1	1
2	2
3	3

KOLUMNA_1	KOLUMNA_3
1	Y
2	N
4	Y

Inner join

W przypadku tego złączenia, wiersze w których wybrana przez nas kolumna się nie pokrywa, zostaną zatracone.

KOLUMNA_1	KOLUMNA_2	KOLUMNA_3
1	1	Y
2	2	N

Left join

W przypadku tego złączenia, wiersze z prawej tabeli w których wybrana przez nas kolumna się nie pokrywa, zostaną zatracone.

KOLUMNA_1	KOLUMNA_2	KOLUMNA_3
1	1	Y
2	2	N
3	3	(NULL)

Right join

W przypadku tego złączenia, wiersze z lewej tabeli w których wybrana przez nas kolumna się nie pokrywa, zostaną zatracone.

KOLUMNA_1	KOLUMNA_2	KOLUMNA_3
1	1	Y
2	2	N
4	(NULL)	Y

Full join

W przypadku tego złączenia, żaden z wierszy nie zostanie zatracony.

KOLUMNA_1	KOLUMNA_2	KOLUMNA_3
1	1	Y
2	2	N
3	3	(NULL)
4	(NULL)	Y

Praktyczne zastosowanie

W skrypcie Józefa Woźniaka możemy znaleźć taki przykład, który zestawia nam imiona zawodników, kluby oraz daty urodzenia kobiet.

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

Unia

Unia łączy wyniki dwóch operacji select w jedną tabelę. Domyślnie powtarzające się wyniki zostaną połączone. Aby do tego nie dopuścić, należy użyć **union all**. Unia może łączyć więcej selectów. Użycie unii jest możliwe, tylko dla tej samej ilości kolumn, tego samego rodzaju.

```
1
2 select COALESCE(t1.KOLUMNA_1, t2.KOLUMNA_1) as KOLUMNA_1, t1.
    KOLUMNA_2, t2.KOLUMNA_3
3 from TABLICA_1 t1
4 full join TABLICA_2 t2 on t1.KOLUMNA_1 = t2.KOLUMNA_1
5 where t2.KOLUMNA_3 = 'N'
6
7 union
8
9 select COALESCE(t1.KOLUMNA_1, t2.KOLUMNA_1) as KOLUMNA_1, t1.
    KOLUMNA_2, t2.KOLUMNA_3
10 from TABLICA_1 t1
11 full join TABLICA_2 t2 on t1.KOLUMNA_1 = t2.KOLUMNA_1
12 where t1.KOLUMNA_1 < 2;
```

KOLUMNA_1	KOLUMNA_2	KOLUMNA_3
2	2	N
1	1	Y

Wykonywanie obliczeń w zdaniach

Tablica Dual

Tablica Dual jest automatycznie tworzona przy postawieniu serwera oracle. Zawiera ona tylko jedną wartość, VARCHAR2 równy "DUMMY". Używana jest na przykład gdy chcielibyśmy wykonać jakieś obliczenia, jednak nie potrzebujemy pobrania żadnych danych z tablic.

```
1 select 2137 + 420 / 69 - 1337 as "Dobry wynik"
2 from dual;
```

Round

round pozwala na zaokrąglanie liczb. Przyjmuje on 2 parametry, liczbę zaokrąglaną oraz miejsce do którego powinna ona być zaokrąglona. Domyślnie zaokrąglanie jest do liczb całkowitych. Możliwe jest ustawienie drugiego parametru na liczbę ujemną, wtedy zacznie on zaokrąglać do liczby 10 * n.

```
1 select round (21.37420, 1)  as "Round to 1"
2        round (21.37420)     as "Round to 0"
3        round (21.37420, -1) as "Round to -1"
4 from dual;
```

Round to 1	Round to 0	Round to -1
21.3	21	20

Concat

Funkcja **concat** łączy dwa ciągi znaków w jeden. Możemy używać ją także dla jej wyników.

```
1  create table tablica_1 ( kolumna_1 number Primary Key,
2                           kolumna_2 varchar2(50),
3                           kolumna_3 varchar2(50),
4                           kolumna_4 varchar2(50)
5  );
6
7  insert into TABLICA_1 values (1, 'pozdrawiam', 'serdecznie', '
       czytelnika');
8  insert into TABLICA_1 values (2, 'życz', 'smacznej', 'kawusi');
9
10 SELECT kolumna_1 as wiersz,
```

```
11          CONCAT(CONCAT(CONCAT(kolumna_2, ' '), kolumna_3), CONCAT
                (' ', kolumna_4)) AS merged_string
12 FROM tablica_1;
```

wiersz	concat
1	pozdrawiam serdecznie czytelnika
2	życzę smacznej kawusi

Warto wiedzieć, że tym konkretnym przypadku możliwym jest osiągnięcie tego samego w znaczny sposób.

```
1 SELECT kolumna_1, kolumna_2 || ' ' || kolumna_3 || ' ' ||
    kolumna_4 AS merged_string
2 FROM tablica_1;
```

Upper, lower oraz initcap

Funkcje **upper** oraz **lower** są wykorzystywane w celu standaryzacji wyników. Jak nazwa wskazuje, zamieniają one ciąg znaków tylko na duże litery, małe litery lub duże pierwsze litery. Stosowanie ich jest konieczne, ponieważ 'KAWA' != 'Kawa' != 'kawa'! , co ma kluczowe znaczenie przy używaniu **where**. Przydaje się także do standaryzacji danych podawanych przez użytkownika, np **upper(:x)**.

```
1 select upper(kolumna_2) as upper, lower(kolumna_3) as lower,
    initcap(kolumna_4) as initcap
2 from tablica_1;
```

upper	lower	initcap
POZDRAWIAM	serdecznie	Czytelnika
ŻYCZĘ	smacznej	Kawusi

Substr

Funkcja substr wycina fragment z podanego łańcucha znaków i zachowuje się dokładnie tak samo jak w C++, podajemy łańcuch znaków, pozycje startową oraz liczba znaków.

```
1 select substr(kolumna_4,0,3) as czykaw?
2 from tablica_1;
```

czykaw?
czy kaw

Operacje na datach

Przechowywanie dat

Wszystkie daty są formatowane w sposób definiowany przez administratora serwera. Zapytanie **select sysdate as Data systemowa from dual;** pozwala nam zobaczyć w jaki sposób przechowywane są one na naszym serwerze. Daty można dodawać i zwiększać o numer.

Jesteśmy w stanie też zmienić sposób przechowywania danych na czas sesji.

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

Ze względu na obowiązkową standaryzacje, musimy używać funkcji **to_date**. Przekazuje ona systemowi dane o tym w jaki sposób chcemy przedstawić date. Widać to na przykładzie ze skryptu:

```
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-ŃWRZESIE-2018', 'DD-MONTH-YYYY' ) + 15 as
             "Koniec"
from dual;
```

Extract

Extract pozwala na pozyskaniu dnia/miesiąca/roku.

```
select extract ( year from sysdate ) as Rok ,
extract ( month from sysdate ) as Miesiac ,
extract ( day from sysdate ) as Dzien
from dual;
```

to_char

to_char zamienia date w tekst. Tak jak wszystko, można go zagnieździć. 'fm' używane jest do usuwania spacji Wymaga on podania przez nas typu formatowania.

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

Funkcje ogólne

Operacje z NULL

funkcja	użycie	rezultat
nvl	x,y	jeśli x jest równe NULL zamienia je na y
nullif	x,y	return x = y ? NULL : x
coalesce	x,y,...,n	zwraca pierwszy != NULL

NVL jest używane, w momencie którym potrzebujemy sumować liczby, a występuje ryzyko pojawienia się nulla.. Jak wiemy ze wcześniejszej rozpiski, NULL + wartość daje NULL, co zapewne nie jest oczekiwanym przez nas wynikiem.

```
1
2  insert into tablica_1 values (1, NULL, 1, 1);
3
4  select kolumna_1, kolumna_2, kolumna_3, kolumna_4,
5         kolumna_2, 0 + kolumna_3, 0 + kolumna_4, 0 as suma
6  from tablica_1;
7
8  -- suma : <NULL>
9
10 select kolumna_1, kolumna_2, kolumna_3, kolumna_4,
11        nvl(kolumna_2, 0) + nvl(kolumna_3, 0) + nvl(kolumna_4,
              0) as suma
12 from tablica_1;
13
14 -- suma : 2
```

Case

Case ... where działa dokładnie tak samo jak w wielu językach programowania. Można go używać w dwóch różnych postaciach. Oczywiście warunki wykonują się od góry do dołu, spełnienie jednego z nich kończy sprawdzanie.

```
1 SELECT
2     CASE
3         WHEN MOD(numer, 3) = 0 THEN 'łza'
4         WHEN MOD(numer, 3) = 1 THEN 'średnia'
5         ELSE 'dobra'
6     END AS "typ kawy",
7     kawa
8 FROM KAWA;
```

typ kawy	kawa
średnia	kawa
dobra	kawa
zła	kawa

Decode

Jeżeli zawsze zamieniamy wartości, kiedy są one sobie równe, możemy użyć także **decode** dla uzyskania takiego samego wyniku.

```
1 select kawa,
2 decode (numer,
3         '1', 'średnia',
4         '2', 'dobra',
5         '3', 'łza') as "typ kawy"
6 from kawa;
```

Grupowanie

Funkcje grupujące

Funkcja	Działanie
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

Zapytania grupujące

group sumuje wyniki z pokrywającą się wartością wybraną przez nas, co najlepiej widać na przykładzie ze skryptu. Dzięki niemu jesteśmy w stanie dowiedzieć się, ilu zawodników jest w jakim klubie.

```
1 select nr_klubu as "Nr klubu",
2 count ( * ) as "Liczba zawodników" from bd3_zawodnicy
3 group by nr_klubu
4 order by nr_klubu
```

Funkcja **group** nie może grupować po kolumnie, którą chcemy agregować. Możemy także grupować także po wielu kolumnach.

having

having jest swego rodzaju odpowiednikiem dla **where** używanego przy grupowaniu. Jeżeli chcemy zgrupować wszystkie wiersze mające numer > 3, użyjemy **group by ... having**. Jeżeli chcemy zobaczyć wszystkie wiersze z numerem większym niż 3, użyjemy **where**. Nie można używać ich naprzemiennie! Najpierw wykona się zawsze **where** a następnie **group by ... having** , więc używając **where** jesteśmy w stanie wybrać interesujące nas zgrupowania.

kolumna_1	kolumna_2	kolumna_3
1	1	1
2	1	5
3	2	9
4	2	1
5	3	3
6	3	9

```
SELECT kolumna_2, AVG(kolumna_3) as average_kolumna_3
FROM test
GROUP BY kolumna_2;
```

kolumna_2	kolumna_3
1	3
2	5
3	6

```
SELECT kolumna_2, AVG(kolumna_3) as average_kolumna_3
FROM test
GROUP BY kolumna_2
HAVING AVG(kolumna_3) > 3;
```

kolumna_2	kolumna_3
2	5
3	6

```
SELECT kolumna_2, AVG(kolumna_3) as average_kolumna_3
FROM test
WHERE kolumna_3 != 1
GROUP BY kolumna_2
HAVING AVG(kolumna_3) > 3;
```

kolumna_2	kolumna_3
1	5
2	9
3	6

Obiekty bazodanowe

Sekwencje

Sekwencje tak samo jak i tablice, tworzy się używając **drop, create, alter**. Przed pierwszym wywołaniem, wartość nie istnieje.

```
1 create sequence SEKWENCJA_1
2 increment by 1
3 start with 1
4 maxvalue 999999999999999999999999999
5 cache 20;
```

Możemy także ręcznie zobaczyć numer w aktualnej sekwencji oraz zwiększyć go.

```
1 select SEKWENCJA_1.nextval from dual;  -- ustawi wartosc na 1 i
      ja wyswietli
2 select SEKWENCJA_1.currval from dual;  -- wyswietli 1
```

Sekwencje takie są przydatne do tworzenia kluczy głównych dla wielu elementów.

```
1 create table KAWA(numer number PRIMARY KEY,
2                   kawa varchar2(4)
3 );
4
5 insert into KAWA values (SEKWENCJA_1.nextval, 'KAWA');
6 insert into KAWA values (SEKWENCJA_1.nextval, 'KAWA');
7 insert into KAWA values (SEKWENCJA_1.nextval, 'KAWA');
8
9 select * from KAWA;
```

wartosc	kawa
1	kawa
2	kawa
3	kawa

Indeks

Indeksy służą do porządkowania tabel oraz pozwala na przyśpieszenie bazy danych, jednak zwiększają ich wielkość. Indeksy tworzą się automatycznie przy tworzeniu klucza głównego. To indeksy, a nie tabele jak mogłoby się wydawać, odpowiedzialne są za pilnowanie unikatowości danych w kolumnach. Indeksy jednak *nie muszą* być unikatowe.

Jest także możliwe tworzenie nowych indeksów ręcznie. Operuje się na nich tak samo jak na innych strukturach, używając **create, drop, alter**.

```
create table test (kolumna_1 number PRIMARY KEY,
                   kolumna_2 number
);

create unique index idx_test on test( kolumna_2);

insert into test values (1, 1);
insert into test values (2, 1);
```

W przypadku zaprezentowanego kodu, spotkamy błąd ORA-00001, mówiący o próbie dodania kopii już istniejącego elementu do indeksu.

Nie trzeba jednak deklarować sekwencji zawsze ręcznie, można użyć opcji unique przy tworzeniu tablicy.

```
create table test (kolumna_1 number PRIMARY KEY,
                   kolumna_2 number unique
);
```

Perspektywy

Perspektywy są wirtualnymi tabelami. Służą one do tworzenia zestawień danych potrzebnych do konkretnych celów, konkretnych osób. Mogą być tworzone używając złożonych zapytań, co pozwala nam na odwoływanie się do nich w przyszłości bez ponownego wywoływania zapytania.

Perspektywy tworzy się w specyficzny sposób.

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

-- wykorzystujac tabele utworzona przy demonstracji dzialania sekwencji
create or replace view KAWA_VIEW ("kawusia") as
    select kawa from KAWA;

select * from KAWA_VIEW;
```

kawusia
kawa
kawa
kawa

Warto zauważyć, że przy tworzeniu KAWA_VIEW zmieniliśmy nazwę kolumny z kawa na kawusia. Można to zastosować dla większej ilości kolumn.

Zapytania zagnieżdżone

W zdaniach z **select** możemy zawrzeć kolejne **select**, aby oprzeć nasze wyszukiwania o jego wynik.

```
1 select * from test
2 where kolumna_3 = (select max(numer) from kawa);
```

Podrzędne zdanie musi zwrócić jedną liczbę, a nie zbiór.

Podzapytania dzielimy na skorelowane i nieskorelowane. - Nieskorelowane, widać na przykładzie powyżej. Otrzymana wartość używana jest dla wszystkich. - Skorelowane, byłoby zależne od zawartości tabeli test. Wymagają one wielokrotnego wywoływania podzapytania, co obciąża serwer.

Przykład skorelowanego podzapytania:

```
1 select nr_zawodnika, nr_klubu
2 from bd3_zawodnicy z
3 where extract ( year from sysdate ) - extract ( year from
     data_urodzenia ) >
4  ( select avg ( extract ( year from sysdate ) - extract ( year
      from data_urodzenia ))
5  from bd3_zawodnicy x
6  where x.nr_klubu = z.nr_klubu );
```

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

Jeżeli podzapytanie będzie zbiorem pustym, wynikiem będzie zbiór pusty.

Operatory mnogościowe

IN

in można interpretować jako **distinct inner join**, jednak w przeciwieństwie do join, możemy dodać mu warunki używając **where**.

```
1 select nazwisko, imie from bd3_zawodnicy
2 where nr_zawodnika in (select nr_zawodnika from bd3_wyniki
     where PUNKTY_GLOBALNE > 20)
3 order by nazwisko;
```

Przykład pokaże imiona i nazwiska wszystkich zawodników, którzy uzyskali więcej niż 20 punktów w zawodach.

ALL oraz ANY

all można stosować tak samo jak **in**, jednak można je przyrównywać używając < , > , <>.

kolumna_1	kolumna_2
1	2
2	3
3	4

```
1 select kolumna_1 from test2
2 where kolumna_1 < all (select kolumna_2 from test2);
3 -- wynikiem bedzie
4 -- kolumna_1
5 --           1
```