zajavka

Notatki - SQL podstawy - cz.3

Spis treści

SQL i podstawowa składnia . . . 1
- Wkładanie danych do tabelek . . . 1
- Odczytywanie danych z tabel . . . 2
  - Aliasy . . . 2
  - Where . . . 2
  - Łączenie warunków . . . 3
  - Operatory . . . 3
- Sortowanie zwracanego wyniku . . . 5
- Ograniczenie ilości zwracanych wierszy . . . 6
- Zwrócenie tylko unikalnych wartości . . . 6
- Grupowanie . . . 6
- UPDATE rekordu w bazie . . . 8
- DELETE rekordu w bazie . . . 8

SQL i podstawowa składnia

Wkładanie danych do tabelek

Wkładanie zostało użyte od słowa INSERT bo to jest słówko kluczowe, które służy do wypełnienia tabel danymi.

```
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (1, 'Aleksander', 'Wypłata', 33, 8791.12, '2018-03-12');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (2, 'Roman', 'Pomidorowy', 43, 7612.12, '2012-01-01');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (3, 'Anna', 'Rosół', 38, 5728.90, '2015-07-18');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (4, 'Urszula', 'Nowak', 39, 3817.21, '2014-12-15');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (5, 'Stefan', 'Romański', 38, 9201.23, '2020-07-14');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (6, 'Jolanta', 'Kowalska', 27, 6521.22, '2012-06-04');
```

Zwróć uwagę, że wpisujemy `INSERT INTO`, później podajemy nazwę tabelki, później w nawiasach podajemy nazwy kolumn, dla których będziemy 'wkładać' dane, a później dodajemy wartości jakie mają się znaleźć w konkretnych wypisanych kolumnach.

Odczytywanie danych z tabel

Dopiero teraz jak mamy już tabelę uzupełnioną danymi to będzie miało jakikolwiek sens żeby próbować te dane odczytać.

```
SELECT * FROM EMPLOYEES;
```

- **SELECT -** tak jak nazwa mówi, komenda mówi o pobieraniu danych,
- *** -** gwiazda oznacza, że mamy pobrać dane ze wszystkich kolumn w tabeli. Możemy również określić jakie konkretnie kolumny mamy zwrócić w rezultacie oddzielając je przecinkiem,
- **FROM -** określa z jakiej tabeli będziemy pobierać dane,

Natomiast jeżeli chcemy wyświetlić dane z konkretnych kolumn, możemy napisać takie query:

```
SELECT
       ID,
       NAME,
       SURNAME
FROM EMPLOYEES;
```

Aliasy

Możemy też przy tym nadać tym kolumnom aliasy. Oznacza to, że tylko w widoku, który wyświetlimy, kolumny te mogą nazywać się inaczej, ale nie zmienia to nic w samej tabeli.

```
SELECT
       ID AS MY_ID,
       NAME AS MY_NAME,
       SURNAME AS MY_SURNAME
FROM EMPLOYEES;
```

Where

Teraz jest o tyle prosto, że mamy 6 wierszy w naszej tabelce, więc możemy pokazać wszystkie wiersze, ale w praktyce w bazach danych są przetrzymywane tysiące wierszy. Możemy zatem mieć potrzebę aby pobrać tylko dane osób, które nazywają się `Roman`.

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman';
```

W ten sposób pobierzemy z bazy tylko rekordy, dla których imię ma wartość `Roman`. Zwróć uwagę, że mamy pojedyncze `=`, a nie `==`.

Łączenie warunków

Warunki podane w `WHERE` możemy ze sobą łączyć za pomocą operatorów `AND` lub `OR`. W takim przypadku zadziała to tak jak z operatorami logicznymi w Javie. Przykładowo:

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman' AND SURNAME = 'Pomidorowy';
```

Ewentualnie:

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman' OR NAME = 'Anna';
```

Operatory

W przypadku operatorów, poruszę również najczęściej używane, nie będą to wszystkie możliwe ☺.

Operatory arytmetyczne

Przykładowe operatory arytmetyczne:

Słownie	Operator	Opis
dodawanie	+	Dodaje do siebie wartości użyte z operatorem
odejmowanie	-	Odejmuje od siebie wartości użyte z operatorem
mnożenie	*	Mnoży przez siebie wartości użyte z operatorem
dzielenie	/	Dzieli przez siebie wartości użyte z operatorem
modulo	%	Dzieli lewy operand przez prawy i zwraca resztę z dzielenia

Przykład użycia:

```
SELECT
       NAME,
       AGE % 10 AS AGE_MOD
FROM EMPLOYEES;
```

Operatory porównania

Operatory te są podobne do tych, które poznaliśmy już w samej Javie. Możemy stosować te operatory przy określaniu jakie warunki mają spełniać dane, które chcemy `SELECTować`. Możemy ich używać przykładowo w warunku `WHERE`. Pamiętajmy, że wynikiem operatorów poniżej jest wartość `true`/`false`.

Operator	Opis	Przykład
=	Operator porównania, sprawdza, czy operandy po obu stronach wyrażenia są sobie równe	SELECT * FROM EMPLOYEES WHERE NAME = 'Roman';
!=	Operator nierówności, sprawdza czy operandy po obu stronach wyrażenia są sobie nierówne	SELECT * FROM EMPLOYEES WHERE NAME != 'Roman';
<>	Operator różności, sprawdza czy operandy po obu stronach wyrażenia są różne, czyli w sumie to samo co poprzedni operator	SELECT * FROM EMPLOYEES WHERE NAME <> 'Roman';
<	Operator mniejszości, sprawdza czy lewa część wyrażenia jest mniejsza niż prawa	SELECT * FROM EMPLOYEES WHERE SALARY < 5000;
>	Operator większości, sprawdza czy lewa część wyrażenia jest większa niż prawa	SELECT * FROM EMPLOYEES WHERE SALARY > 5000;
<=	Operator mniejsze-równe, sprawdza czy lewa część wyrażenia jest mniejsza lub równa prawej	SELECT * FROM EMPLOYEES WHERE SALARY <= 5000;
>=	Operator większe-równe, sprawdza czy lewa część wyrażenia jest większa lub równa prawej	SELECT * FROM EMPLOYEES WHERE SALARY >= 5000;

Operatory logiczne

Poznaliśmy już wcześniej operatory logiczne jako `&&` lub `||`. Tutaj mamy trochę więcej możliwości niż w samej Javie. Wcześniej w notatce wspomniane zostały już 2 operatory `OR` oraz `AND`. Pamiętajmy, że wynikiem operatorów poniżej jest wartość `true`/`false`.

Operator	Opis	Przykład
OR	Operator LUB (alternatywa), używany przykładowo w klauzuli WHERE	SELECT * FROM EMPLOYEES WHERE NAME = 'Roman' OR NAME = 'Agnieszka';
AND	Operator I (koniunkcja), używany przykładowo w klauzuli WHERE	SELECT * FROM EMPLOYEES WHERE NAME = 'Roman' AND SURNAME = 'Romański';
IN	Operator sprawdzający, czy wartość w kolumnie jest równa jednej z podanych wartości. Działanie w przykładzie jest analogiczne do przykładu w operatorze OR	SELECT * FROM EMPLOYEES WHERE NAME IN ('Roman', 'Agnieszka');
LIKE	Operator działający podobnie do String.contains()	SELECT * FROM EMPLOYEES WHERE NAME LIKE '%Ro';
BETWEEN	Operator sprawdzający, czy wartość w kolumnie jest zawarta w przedziale podanym przy operatorze	SELECT * FROM EMPLOYEES WHERE AGE BETWEEN 20 AND 30;
IS NULL	Operator sprawdzający, czy wartość w kolumnie jest NULL	SELECT * FROM EMPLOYEES WHERE AGE IS NULL;

Operator	Opis	Przykład
NOT	Operator odwracający znaczenie innych operatorów	SELECT * FROM EMPLOYEES WHERE NAME NOT IN ('Roman', 'Agnieszka');

LIKE - Lubię to

Operator `LIKE` specjalnie wyciągam pod oddzielny fragment ze względu na to, że jest często używany.

`LIKE` działa podobnie do `String.contains()`, ale należy przy tym pamiętać o znaku charakterystycznym `%`. Oznacza on brak znaku albo jeden lub więcej dowolnych znaków. Przykładowo:

Znajdź rekordy, gdzie imię zaczyna się od dowolnych znaków ale kończy się znakami `RO`:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE '%Ro';
```

Znajdź rekordy, gdzie imię zaczyna się od `Ro`, ale kończy się dowolnymi znakami:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE 'Ro%';
```

Znajdź rekordy, gdzie imię ma w środku `Ro`, może zaczynać i kończyć się dowolnymi znakami. Inaczej mówiąc, dopiero ten zapis odzwierciedla metodę `String.contains()`:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE '%Ro%';
```

Sortowanie zwracanego wyniku

Wynik zwracany możemy posortować po konkretnej kolumnie, albo nawet po kilku.

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE DESC;
```

Powyższe zapytanie zwróci nam rekordy z tabeli `EMPLOYEES` posortowane po wieku malejąco. Jeżeli chcielibyśmy posortować te wiersze rosnąco, to albo zamiast `DESC` możemy napisać `ASC`, albo napisać to tak:

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE;
```


Domyślnie sortowanie odbywa się rosnąco, dlatego nie ma potrzeby pisać `ASC`.

Możemy również posortować wynik po kilku kolumnach w kolejności:

```
SELECT *
FROM EMPLOYEES
ORDER BY SALARY DESC, AGE ASC;
```

Ograniczenie ilości zwracanych wierszy

W PostgreSQL do tego służy słówko kluczowe `LIMIT`. Wspominam tutaj o PostgreSQL, bo inne bazy mogą mieć to zrealizowane w inny sposób. Poniższe zapytanie zwróci nam tylko 2 wiersze posortowane domyślnie.

```
SELECT *
FROM EMPLOYEES
LIMIT 2;
```

Natomiast jeżeli interesowałoby nas zwrócenie 5 najmłodszych pracowników, moglibyśmy napisać to tak:

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE ASC
LIMIT 5;
```

Zwrócenie tylko unikalnych wartości

Wyobraźmy sobie, że potrzebujemy zwrócić tylko unikalne wartości jakie występują w danej kolumnie. Przykładowo chcemy się dowiedzieć jakie imiona ludzi występują wśród pracowników naszej firmy. Do tego służy słówko `DISTINCT`:

```
SELECT DISTINCT NAME
FROM EMPLOYEES;
```

Grupowanie

Zanim poruszymy grupowanie to musimy wspomnieć o funkcjach agregujących. Jest to nic innego jak funkcja która z kliku elementów w jakiś sposób zwróci jakąś jedną wartość. Przykładowo może być to wartość maksymalna, minimalna, suma wartości, średnia itp.

Poruszymy takie funkcje agregujące:

Funkcja	Działanie
COUNT	Zlicza ilość elementów w zbiorze

Funkcja	Działanie
SUM	Sumuje wartości elementów w zbiorze
AVG	Wylicza średnią wartość elementów w zbiorze
MIN	Określa wartość minimalną dla elementów w zbiorze
MAX	Określa wartość maksymalną dla elementów w zbiorze

Funkcje powyżej mogą być wykonywane bez klauzuli `GROUP BY`, która jest poruszana poniżej, przykładowo:

```
SELECT
    COUNT(AGE),
    SUM(AGE),
    AVG(AGE),
    MIN(AGE),
    MAX(AGE)
FROM EMPLOYEES;
```

Znając już funkcje agregujące możemy przejść do klauzuli `GROUP BY`. Pamiętasz ze Streamów w programowaniu funkcyjnym, że mieliśmy możliwość pogrupowania obiektów po jakiejś wartości i otrzymywaliśmy wtedy mapę `klucz:lista_wartości`? Tutaj jest podobnie. Wyobraźmy sobie, że chcemy pogrupować rekordy po wieku. Otrzymalibyśmy wtedy mapę `wiek:lista_ludzi_w_tym_wieku`. Natomiast z racji, że przedstawiamy dane w tabelce, to musimy taki zapis wepchnąć do jednego wiersza. Przykładowo zapytanie poniżej nie zostanie wykonane poprawnie. Musimy określić funkcję agregującą te listy ludzi dla danego wieku.

```
SELECT *
FROM EMPLOYEES
GROUP BY AGE;
```

Może pojawić się teraz pytanie, czy możliwe jest ominięcie tej agregacji i przedstawienie w tabeli mapy, która została wspomniana w taki sposób, żeby było widać całą listę dla klucza, tak jak poniżej:

AGE	NAME
33	Aleksander, Roman, Stefan
28	Agnieszka, Karol, Michał
34	Anna, Urszula, Jolanta

Od razu odpowiadam, jest to możliwe, ale o wiele trudniejsze niż poziom, którego uczymy się teraz. Dlatego skupiamy się na funkcjach agregujących.

Zapytanie poniżej zliczy nam ile jest osób w każdym wieku. Najpierw grupujemy osoby w danym wieku `GROUP BY`, dostajemy wtedy mapę `wiek:lista_ludzi_w_tym_wieku`. Następnie wykorzystujemy funkcję `COUNT`, aby zliczyć rozmiary tych list i przedstawić mapę `wiek:ilość_ludzi_w_tym_wieku` w formie tabelki. Możemy w tym celu wykorzystywać również inne funkcje agregujące.

```
SELECT AGE, COUNT(AGE)
FROM EMPLOYEES
GROUP BY AGE;
```


UPDATE rekordu w bazie

Rekordy w bazie danych mogły być tworzone od zera, ale bardzo często zdarzy się, że taki rekord będziemy musieli zaktualizować. Przykładowo możemy napisać, żeby od dzisiaj wszystkie `Anny` w naszej firmie zarabiały `10000` pieniędzy.

```
UPDATE EMPLOYEES
SET SALARY = 10000
WHERE NAME = 'Anna';
```

Jak widzisz używamy słowa kluczowego `UPDATE`, a następnie określamy jakie pola chcemy zaktualizować. Ważne też jest aby pamiętać o klauzuli `WHERE` inaczej zaktualizujemy wypłatę dla wszystkich pracowników.

A co jeżeli chcielibyśmy zaktualizować jednocześnie dane w kilku kolumnach? Niech każdy `Roman` ma na nazwisko `Zajavkowy` i ma `20` lat.

```
UPDATE EMPLOYEES
SET SURNAME = 'Zajavkowy', AGE = 20
WHERE NAME = 'Roman';
```

DELETE rekordu w bazie

Dane możemy również z bazy usuwać. Należy jednak pamiętać ponownie, aby nie skasować danych z całej tabeli jednocześnie. Jeżeli pominiemy klauzulę `WHERE`, usuniemy wszystkie dane z tabeli.

```
DELETE
FROM EMPLOYEES
WHERE ID = 5;
```