Zajęcia 2: Wykład (SQL cz. 2)

Date: 2015-10-08
tags: zaj2, wykład, materiały
category: materiały

> **Note**
> Wykład do pobrania również w wersji PDF.

Spis treści

Rzeczy do zapamiętania

W praktyce (poza zapytaniami `SELECT`) bardzo rzadko ręcznie pisze się kod SQL. Istnieją narzędzia które automatyzują wszystkie żmudne zadania związane z bazą danych, jednak bardzo przydatną umiejętnością jest **czytanie** kodu SQL, znacznie ułatwia ono np. poszukiwanie błędów.

Celem tych zajęć jest to żebyście **z grubsza** umieli przeczytać kod SQL generowany przez Django, nie oczekuje że napiszecie w trakcie tych zajęć jedno polecenie `CREATE TABLE`.

Relacje w relacyjnej bazie danych

Na razie w tabeli SQL przechowywaliśmy bardzo proste dane, wszystkie dane były przechowywane w jednej tabeli do której odnosiły się zapytania. Schemat taki jest bardzo prosty i nie wykorzystuje w pełni możliwości baz relacyjnych, by je wykorzystać musimy móc wyrazić relacje pomiędzy tabelami.

W bazie danych możliwe są dwa podstrawowe rodzaje relacji:

- Relacje jeden-do-wielu
- Relacje wiele-do-wielu

By zrozumieć jak są one zaimplementowane musimy przypomnieć sobie pojęcie *klucza głównego* i *klucza obcego*.

klucz główny *z ang.* primary key

> To kolumna (lub zestaw kolumn) która jednoznacznie identyfikuje dany wiersz, spełnia ona dwa wymagania:
>
> - Wartości w tej kolumnie są unikalne dla każdego wiersza.
> - Wartości w tej kolumnie nigdy nie przyjmuje wartości `NULL`.

By jednoznacznie zidentyfikować wiersz w tabeli, starczy znać wartość jego klucza głownego.

Relacja jeden-do-wielu

Powiedzmy, że mamy w bazie danych tabelę `student`, która zawiera syntetyczny klucz główny w kolumnie `id`. Chcemy do tej bazy danych dodać informację o ocenach studentów.

Tabela student wygląda tak:

student	
id	**name**
1	Jan Kowalski
2	Józef Nowak
3	Karol Alfons

Pierwszy pomysł jest taki, stwóżmy tabelę `OCENA` która wygląda tak:

OCENA		
id	**student_id**	**mark**
1	1	2
2	1	3
3	2	4

Kolumna `student_id` oznacza identyfikator studenta, który otrzymał daną ocenę, relatywnie łatwo jest budować zapytania do takiego schematu (o tym jak się to robi powiem później).

Zasadniczo jest jeden problem z takim schematem, co się stanie jeśli ktoś wprowadzi do kolumny `student_id` wartość np. 500 (której nie ma w kolumnie `id` tabeli `student`)? Jeśli pojawi się taka wartość wiele zapytań nie będzie działać poprawnie.

By zapobiec temu problemowi można wprowadzić do bazy danych ograniczenie zwane `kluczem obcym`, ograniczenie to gwarantuje że jeśli w kolumnie `student_id` jest wartość X to istnieje wiersz w tabeli `student` o `id` równym X.

Klucz obcy robi dwie rzeczy:

- Nie uda się umieścić w tabeli `OCENA` wiersza o `student_id` łamiącym ograniczenie.
- Przy usuwaniu/zmianie wierszy z tabeli `student` domyślnie baza danych usunie wszystkie odpowiadające danemu studentowi oceny (zachowanie to jest konfigurowalne).

Note

Uwaga: poprawna implementacja klucza obcego, działająca w przypadku równoległej edycji tabeli przez wielu użytkowników jest **nietrywialna**.

Note

Naturalny klucz główny (*z ang.* natural key), to klucz główny, na który składają się kolumny już istniejące w bazie danych mające znaczenie w *świecie rzeczywistym*.
Przykładowo w tabeli przechowującej studentów możemy uznać, że dobrym kluczem głównym będzie unikalny numer PESEL.
Syntetyczny klucz główny (*z ang.* synthetic key), to klucz głowny, którego wartość ma znaczenie tylko wewnątrz bazy danych, i została przez nią przypisana. Praktycznie zawsze syntentyczne klucze głowne generuje się za pomocą "sekwencji", tj. są one przyznawane "po kolei".
Przykładowo uczelnia przyznaje studentom syntetyczne identyfikatory (nr. indeksu).
Według wielu administratorów w zasadzie zawsze należy dodawać do tabeli klucz syntetyczny. Ma on takie zalety:

- Jego wartość nigdy się nie zmienia (zmianę wartości w klucza naturalnego może wymusić zmiana w świecie).
- Nie zależy od zachowania świata zewnętrznego.
- Klucze sztuczne są mniejsze, generalnie są przechowywane jako 8 (czasem 16) bitowy typ stałoprzecinkowy (potocznie: `long` lub `long long`).
- Joiny po kluczach sztucznych mogą być szybsze (sztuczne klucze główne są mniejsze)

Przy naturalnych kluczach głównych łatwo jest przapić nieoczywiste relacje w świecie rzeczywistym, które "psują" założenia (np. student z Ukrainy nie musi mieć przyznanego numeru pesel).

Relacja wiele-do-wielu

Powiedzmy że do naszego schematu chcemy dodać informację o tym na jakie przedmioty student się zapisał. Jeden student może zapisać się na wiele przemdiotów, no i oczywiście w jednym przedmiocie bierze udział wielu studentów. Takiej relacji nie da się zaimplementować za pomocą pojedyńczego klucza obcego.

Tabela student wygląda tak:

student	
id	**name**
1	Jan Kowalski
2	Józef Nowak
3	Karol Alfons

Tabela kurs wygląda tak:

course	
id	**name**
1	Programowanie
2	Fizyka
3	Underwater basket weaving

By zaimplementować relację wiele-do-wielu między tymi tabelami, musimy stworzyć nową tabelę `student_course`, tabela ta będzie miała klucze obce, zarówno do tabeli `course` jak i do tabeli `student`.

student_course	
student_id	**course_id**
1	1
1	2
1	3
2	3

Student o `id` 1 uczestniczy w kursie o id `2` jeśli w tabeli `student_course` jest wiersz o wartości `student_id` równej 1 oraz `course_id` równej 2.

> **Note**
> Kolumna `student_id` jest kluczem obcym to tabeli `student`, a kolumna `course_id` kluczem obcym to tabeli `course`.
> Dodatkowo

Indeksy

Rozważmy tabelę:

Tabela student wygląda tak:

student	
id	**name**
15	Jan Kowal
1	Jan Kowalski
10	Józef Nowak
...	...
500000	Karol Alfons

Dane nieposortowane

Naszym zadaniem jest znaleźć imie i nazwisko studenta o `id` równym 234, ile czasu zajmie nam (średnio) znalezienie tego studenta w funkcji ilości rekordów w bazie danych?

W tak postawionym problemie średnio należy sprawdzić $\frac{N}{7}2$ rekordów zanim znajdziemy ten o odpowiednim ID.

Dane posortowane

Ile czasu zajmie odnalezienie studenta jeśli dane w tabeli są posortowane względem indeksu? W tym przypadku będzie trzeba sprawdzić $\log_2 N$ rekordów.

> **Note**
>
> Można do tego wykorzystać algorytm zwany binarnym przeszukiwaniem.
> Algorytm ten opiera się na następującej obserwacji, weźmy element E znajdujący się w środku tabeli, jego `id` może być:
>
> - Równe poszukiwanemu --- wtedy problem jest rozwiązany
> - Mniejsze od poszukiwanego --- wtedy wszystkie elemementy znajdujące się przed E również mają `id` mniejsze od poszukiwanego więc można je wykluczuć.
> - Większe od poszukiwanego --- wtedy wszystkie elemementy znajdujące się za E również mają `id` większe od poszukiwanego więc można je wykluczuć.

Indeksy

Przechowywanie posortowanych danych w bazie jest niepraktyczne, główne powody to:

- Konieczność utrzymywania kilku uporządkowań na raz. Chcielibyśmy zarówno móc szybko wyszukiwać studenta znając jego `id`, numer pesel jak i imię.
- Koszt utrzymania sortowania jest duży --- jeśli okaże się że nowy element trzeba wstawić na początku tabeli wszystkie kolejne elementy trzeba przesunąć.

Użyto więc innego rozwiązania: do kolumny można dodać indeks, indeks pozwala szybciej wyszukiwać dane w tabeli jeśli przeszukujemy tabelę z użyciem zindeksowanych kolumn. Istnienie indeksów spowalnia proces dodawania danych do tabeli.

> **Note**
>
> Istnieje dużo typów indeksów, i każdy typ ma inne zastosowanie, detale jednak przekraczają zakres tego przedmiotu.
> Istnieją też indeksy obejmujące wiele kolumn.

Warning

Indeksy nie są za darmo, jeśli mamy tabelę z kolumnami `a`, `b` i `c`, to bez indeksów:

- Przeszukanie (np. za pomocą `SELECt * FROM T WHERE a=3`) tabeli, będzie wymagało odczytania całej tabeli, więc ma złożoność `O(n)`.
- Dodanie wiersza do tabeli zajmie zawsze tyle samo czasu `O(1)`.

Jeśli dodamy indeks na kolumnie `a`, to:

- Przeszukanie zajmie nam `O(log(n))`.
- Dodanie wiersza zajmie również `O(log(n))`.

Dodatkowo indeksy powodują, że rozmiar tabeli rośnie, indeks na kolumnie `a` ma rozmiar rzędu rozmiaru kolumny `a`.

Jeśli w tabeli `T` kolumna `id` jest kluczem głownym baza danych Posgresql tworzy na niej indeks automatycznie.

Wybieranie daych o relacjach

Aliasy

Przy wybieraniu danych z tabeli możemy nadać tabeli alias, tj zamiast napisać:

```
SELECT id from student;
```

możemy napisać:

```
SELECT s.id FROm student AS s;
```

Sformuowanie `student AS s` oznacza, że w dalszej części zapytania do tabelki `student` można odwoływać się poprzez alias `s`, a wyrażenie `s.id` oznacza kolumnę `id` z tabeli student.

Wybieranie danych z wielu tabel

Gdy na liście `FROM` zapytania jest wiele tabel powoduje to wybranie danych z **kartezjańskiego produktu** wierszy tych tabel.

Jeśli mam tabele:

student	
id	**name**
1	Jan Kowalski
2	Józef Nowak
3	Karol Alfons

oraz:

course	
id	**name**
1	Programowanie
2	Fizyka
3	Underwater basket weaving

Zapytanie:

```
SELECT s.id, c.id from student AS s, course AS c;
```

zwróci taki zestaw danych:

s.id	**c.id**
1	1
2	1
3	1
1	2
2	2
3	2
1	3
2	3
3	3

> **Note**
> Oczywiście dla dużych tabel nikt nie wybiera takiego kartezjańskiego produktu, ale bardzo łatwo jest z takiego zestawu za pomocą odpowiedniej klauzuli `WHERE` wybrać np. kursy danego studenta.

By wybrać informację o średniej dla każdego studenta możemy wykonać takie zapytanie:

```
SELECT s.id, AVG(m.mark)
  FROM
    student as s,
    mark as m
  WHERE s.id = m.student_id
  GROUP BY s.id
  ORDER BY s.id;
```

W zapytaniu tym:

- Wybieramy dane z dwóch tabeli `student` oraz `mark`, dodatkowo dodajemy do tych tabeli aliasy.
- Za pomocą klauzuli WHERE do wybieramy tylko oceny dla danego studenta.
- Wybieramy średnią ocenę dla każdego studenta.

Wybieranie danych za pomocą operatora `JOIN`

Bardzo podobne efekty można uzyskać za pomocą operatora `JOIN`, poniższy przykład będzie dawał dokładnie te same wyniki co poprzedni:

```
SELECT s.id, AVG(m.mark)
  FROM
    student as s
  JOIN mark as m ON s.id = m.student_id
  GROUP BY s.id
  ORDER BY s.id;
```

Rozważmy jeszcze jeden problem: powiedzmy że chcemy wypisać listę studentów, oraz ich średnią, *ale część studentów nie posiada jeszcze żadnych ocen* w takim wypadku powyższe zapytanie ich pominie, co w naszym przypadku jest niepożądane.

Rodzaje JOINÓW

W postgresql jest kilka rodzajów `JOIN` ów:

- Zwykły join `INNER JOIN`, `CROSS JOIN`, `JOIN`, jest równoważny wyrażeniu `FROM table1, table2`, wybiera kartezjański produkt wierszy z obydwu tabel ograniczony pewnymi warunkami.
- `LEFT OUTER JOIN` podobnie jak join samo jak `JOIN`, ale gwarantuje że w wyniku zapytania będzie obecny każdy wiersz z tabeli *po lewej stronie operatora JOIN*

  Implementacja `LEFT OUTER JOIN` działa następująco: jeśli jakiś wiersz z lewej tabeli byłby usunięty z tego powodu, że nie ma odpowiadających mu wierszy tabeli z prawej strony, to i tak jest dodawany do zbioru wynikowego, ale przypisuje zakładamy że wszystkie kolumny tabeli z prawej strony przypisane do tego wiersza będą miały wartość `NULL`.

  Note
  W zapytaniu zawierającym: `SELECT * FROM student as s LEFT OUTER JOIN mark...` tabelą "po lewej stronie operatora join" jest tabela `student`.
- `RIGHT OUTER JOIN` działa tak samo jak `LEFT OUTER JOIN`, ale dla tabeli *po prawej stronie operatora join*.
- `OUTER JOIN` działą tak samo jak `LEFT OUTER JOIN`, ale dla obydwu tabel.

By wyświetlić również wiersze dla studentów bez ocen, należy zatem wykonać zapytanie:

```
SELECT s.id, AVG(m.mark)
  FROM student as s LEFT OUTER JOIN mark as m ON s.id = m.student_id
  GROUP BY s.id
  ORDER BY s.id;
```

Tworzenie schematu bazy danych (opcjonalne)

Pojęcie bazy danych

System zarządzania bazami danych Postgresql, pozwala na jednym komputerze zarządzać wieloma bazami danych, do tworzenia baz danych można użyć programu `pgadminIII`, albo polecenia CREATE DATABASE.

Bazy danych są od siebie całkowicie odseparowane, "nie widzą" swoich tabel itp.

Dobrą praktyką jest trzymanie oddzielnych projektów (u nas: każdych zajęć) w oddzielnej bazie danych.

Użytkownicy w bazie dancyh postgresql

Domyślnie w bazie danych postgresql zainstalowany jest jeden użytkownik superadministrator o nazwie `postgres`.

By stworzyć nowego użytkownika należy wykonać polecenie: CREATE USER, lub za pomocą `pgAdminIII`

> **Note**
> Na Windowsie użytkownikowi należy podać hasło, na Linuksie można skorzystać z `peer authentication`, w której użytkownik zalogowany w systemie operacyjnym jako użytkownik `foo` zostanie zalogowany jako użykownik `foo` w bazie danych (jeśli użytkownik o takiej nazwie w bazie danych istnieje).

Tworzenie tabel

> **Note**
> Celem tych zajęć jest to żebyście **z grubsza** umieli przeczytać kod SQL generowany przez Django, nie oczekuje że napiszecie w trakcie tych zajęć jedno polecenie `CREATE TABLE`.

> **Note**
> Polecam tworzyć tabele za pomocą interfejsu administracyjnego `pgadmin3`. Jest szybciej niż przez konsolę.

Definicja tabeli w postgresql składa się z:

- Listy kolumn
- Ograniczeń
- Indeksów
- triggerów (o tym nie powiemy)
- Zasad (o tym nie powiemy)

- Uprawnień (o tym nie powiemy)
- i innych rzeczy

Do tworzenia tabel służy klauzula:

```
CREATE TABLE "FOO"
(
    [lista kolumn, indeksów, ograniczeń i triggerów , może być pusta]

);
```

Typy kolumn

`character varying` Ciąg znaków o zmiennej długości. Uwaga: większość baz danych wymaga podania maksymalnej ilości znaków w takim typie, postgres natomiast tego nie wymaga.

`TEXT` Praktycznie odpowiednik `character varying`.

`smallint, integer, and bigint` Liczby całkowite różnych rozmiarów

`real, double precision` Liczba zmiennoprzecinkowa o ustalonej dokładności 64bity. Dokładność tych liczb jest taka jak systemu operacyjnego.

`numeric` Liczba stałoprzecinkowa.

W telegraficznym skrócie: *zwykłe* liczby zmiennoprzecinkowe mają pewne niedokładności, a pewne cechy ich zachowania nie są do końca określone (zależą od infrastruktury procesora).

Przykładowo dla liczb zmiennoprzecinkowych (`floating point` możliwe jest takie działanie:

```
>>> 0.2 + 0.1
0.30000000000000004
```

(wynika to z problemów zaokrągleń). Liczby stałoprzecinkowe mają dobrze zdefiniowane zasady zaokrąglania, co jest przydatne w bazach danych będących backendem np. do systemów księgowych.

Dokładne wyjaśnienie na na wikipedii oraz w podręczniku postgresql.

`date` Dzień, miesiąc i rok.

Umieszczanie dat:

```
date '2001-09-28'
```

`time` Czas (minuta i godzina) z dokładnością do milisekundy

`timestamp` Data i godzina (dokładność do milisekundy)

`timestamp with timezone` Data i godzina (dokładność do milisekundy), z określeniem strefy czasowej.

`serial` Wartości sztucznych kluczy głównych muszą być generowane przez bazę danych.

Najprostszą metodą generowania kluczy głównych jest użycie typu `SERIAL` do kolumny oznaczającej klucz główny:

```
CREATE TABLE "STUDENT_2"
(
    id serial NOT NULL,
    CONSTRAINT "STUDENT_2_pkey" PRIMARY KEY (id )
)
```

Teraz kolejnym wstawianym wierszom kolumny `id` będą automatcznie przypisywane kolejne liczby naturalne.

Definiowanie kolumn

Definicja kolumny w najprostszej postaci jest taka:

```
nazwa_kolumny typ;
```

Na przykład:

```
CREATE TABLE "FOO"
(
    pk integer
);

ALTER TABLE "FOO" ADD COLUMN pk integer;
```

Dodawanie kolumn

```
ALTER TABLE "FOO" ADD COLUMN ....;
ALTER TABLE "FOO" DROP COLUMN nazwa;
ALTER TABLE "FOO" RENAME COLUMN nazwa1 TO nazwa2;
```

Usuwanie tabel

```
DROP TABLE "FOO":
```

Domyślne wartości

Do każdej kolumny możemy dodać domyślną wartość, tj. wartość która będzie przypisana do kolumny, jeśli w poleceniu `INSERT` dana kolumna nie będzie określona.

Klauzula default może określać wartość domyślną jako stałą, lub np. wynik wywołania funkcji.

Klauzula default nie umożliwia odnoszenia się do pozostałych kolumn w danym wierszu (taka funkcjonalność możliwa jest do osiągnięcia za pomocą triggera).

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'), -- default j
    name text,
    price numeric DEFAULT 9.99 -- stałe default
);
```

Klucze obce

By jedna tabela odnosiła się do innej musimy dodać kolejne ograniczenie, tzw. klucz obcy.

Powiedzmy że tabele `student` oraz `mark` mają następującą definicję:

```
CREATE TABLE student
(
  id integer NOT NULL DEFAULT nextval('zaj2_schema_app_student_id_seq'::regclas
  name character varying(100) NOT NULL,
  CONSTRAINT zaj2_schema_app_student_pkey PRIMARY KEY (id)
)

CREATE TABLE mark
(
  id integer NOT NULL DEFAULT nextval('zaj2_schema_app_mark_id_seq'::regclass),
  mark smallint NOT NULL,
  course_id integer NOT NULL,
  student_id integer NOT NULL,
  CONSTRAINT zaj2_schema_app_mark_pkey PRIMARY KEY (id)
)
```

By poinformować silnik bazy danych o tym, że kolumna `student_id` jest kluczem obcym do tabeli student należy wykonać:

```
ALTER TABLE ADD CONSTRAINT 'student_fk' FOREIGN KEY (student_id)
    REFERENCES student (id);
```

Cascade (opcjonalne)

Silnik bazy danych nie pozwoli na wstawienie rzędu danych do tabeli `mark`, jeśli w tym rzędzie będzie odniesienie do nieistniejącego studenta. Jednak co się stanie jeśli już po utworzeniu wiersza w tabeli `mark` usuniemy studenta, do którego dany wiersz się odnosi?

Ponieważ serwer wymusza prawdziwość ograniczeń zawsze, pod koniec transakcji (czym są transakcje powiemy później) baza danych zgłosi wyjątek, że ograniczenie jest niespełnione i zmiany zostaną wycofane.

W dalszej cześci zakładamy że usuwamy rząd z tabeli `student` do którego donosi się jakiś wiersz z tabeli: `mark`.

Zachowanie to można konfigurować, by zobaczyć jak można to zrobić, poażę pełną składnię tworzenia klucza obcego:

```
ALTER TABLE ADD CONSTRAINT 'student_fk' FOREIGN KEY (student_id)
    REFERENCES student (id)
    ON UPDATE NO ACTION ON DELETE NO ACTION;
```

Dokładniej rozszyfrujmy linijkę:

`ON UPDATE NO ACTION ON DELETE NO ACTION.`

Linia ta pozwala wybrać akcję do wykonania przez serwer, gdy zdalny wiersz (w naszym przykładzie zdalny rząd to wiersz z tabeli `student` do którego odnosi się jakaś `mark`), danych jest usuwany (`ON DELETE`) bądź zmieniany (`ON UPDATE`).

Akcje do wybrania są takie:

`NO ACTION` spowoduje nie wykonanie żadnej akcji, co może spowodować wyrzucenie wyjątku podczas zamykania transakcji (nie spowoduje go jeśli potem usuniemy również wiersz ze wszystkuch tabel posiadających klucz obcy do tego wiersza).

`RESTRICT` spowoduje wyrzucenie wyjątku od razu!

`SET NULL` spowoduje ustawienie wartości NULL w kolumnach odnoszących się do kasowanego lub zmienianego wiersza.

`SET DEFAULT` spowoduje ustawienie domyślnej wartości w kolumnach odnoszących się do kasowanego lub zmienianego wiersza

`CASCADE` jeśli zdalny wiersz jest kasowany spowoduje skasowanie wierszy, które się do niego odnoszą, jeśli jest zmieniany spowoduje zmianę wartości w tej tabeli by ciągle odnosiły się do tego samego wiersza.