
Helion
JOE CELKO
PRAKTYKI MISTRZA SQL
PROGRAMOWANIE ZAAWANSOWANE
MK
MORGAN KAUFMANN

Tytuł oryginału: Joe Celko's SQL for Smartie: Advanced SQL Programming, 5th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-2530-2

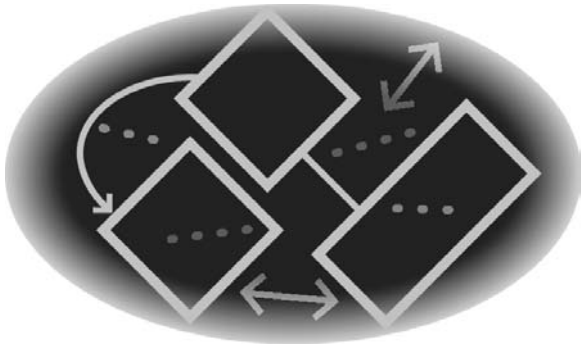


Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/pmsqlp.zip*

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/pmsqlp*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.



# Spis treści


# Błędy, których należy unikać w instrukcjach DDL

POZNAŁEŚ JUŻ podstawowe narzędzia do budowania instrukcji DDL potrzebnych w schemacie. Jednak sama znajomość narzędzi nie oznacza jeszcze, że wiesz, jak się nimi posługiwać. Możesz zapytać o to każdego, kto próbował samodzielnie zajmować się hydrauliką.

## 9.1. Stosowanie skrótu „tbl" i powiązane błędy

W czasach kart dziurkowanych i taśm magnetycznych identyfikatory były krótkie i musiały obejmować metadane dla prostych kompilatorów i systemów wykonujących powtarzalne instrukcje. Klasycznym przykładem były języki FORTRAN I i FORTRAN II. Były one oparte na algebrze i używały instrukcji `GOTO` do zarządzania przepływem sterowania. Występowały w nich tylko dwa liczbowe typy danych — `INTEGER` i `REAL`. Nazwy zmiennych miały do sześciu znaków, musiały rozpoczynać się literą i być unikatowe w programie. Deklarowano je, stosując je w instrukcji, a kompilator musiał wywnioskować typ danych na podstawie nazwy. Zmienne rozpoczynające się od liter od I do N reprezentowały liczby całkowite (typ `INTEGER`), wszystkie pozostałe nazwy oznaczały liczby zmiennoprzecinkowe (typ `REAL`).

Metadane w przedrostkach nazw elementów danych występowały także w systemach operacyjnych, gdzie sprzęt (na przykład napęd taśmowy lub dyskowy) miał zapisane na stałe nazwy w postaci skrótów literowych lub liczbowych. Nie istniały wtedy nowoczesne kompilatory i nie rozdzielano abstrakcyjnej postaci programu od jego implementacji.

Jednak także obecnie stosowane są przedrostki określające typy danych, na przykład `vch_` (typ `VARCHAR(n)`), `int_` (typ `INTEGER`) itd. Angielskie określenie *tibble* (czyli stosowanie skrótu `tbl`) pochodzi od przedrostka `tbl_` w nazwach tabel. Czasem pojawiają się też przedrostki `vw_` (od *Volkswagen Views*, czyli widoki firmy Volkswagen) i jeszcze gorsze nazwy.

## 9.1.1. Standard ISO-11179

Obecnie dostępne są standard ISO-11179 i standardy rozwijane przez jednostkę DM32.8 Meta Data w organizacji *INCITS*. Zgodnie z tymi standardami należy *nazywać rzeczy zgodnie z ich naturą*. Oto co to oznacza:

1. Nie należy podawać typów danych (czyli sposobu fizycznego zapisu danych).
2. Nie należy stosować przedrostków `pk_` ani `fk_` określających sposób używania danego elementu w tabeli (jako klucza głównego lub obcego).
3. Nie należy stosować nazw w formacie „tabela-kolumna”, określających lokalizację danego wystąpienia elementu danych.
4. Tabele to zbiory, dlatego jako ich nazwy należy stosować rzeczowniki zbiorowe lub w liczbie mnogiej. To oznacza, że nazwa `Pracownik` jest zła. `Pracownicy` to lepsza wersja, natomiast właściwe określenie to `Personel`. Podobnie `Drzewo` to zły wybór, `Drzewa` to lepsze rozwiązanie, a `Las` to poprawna forma. Zapewne już rozumiesz ten wzorzec.
5. Nazwa elementu danych może obejmować nazwę atrybutu, podkreślenie i cechę atrybutu. Jeśli ten sam element występuje w jednej tabeli wielokrotnie, jego nazwę można poprzedzić nazwą roli.

Zgodnie ze standardem ISO-11179 nazwy elementów danych powinny mieć format `[<rola>_]<atrybut>_<cecha>`. W przykładzie z pierwotnego dokumentu opisano atrybut `tree` z cechami takimi jak `tree_diameter`, `tree_species` i tak dalej. Niektóre cechy nie dotyczą niektórych atrybutów. Na przykład nie trzeba reprezentować cechy `employee_diameter`, a cecha `employee_species` może okazać się obraźliwa. Cechy atrybutów związane z wartościami ze skal potencjalnie można wykorzystać w tabelach wyszukiwania. Poniżej znajdziesz listę i definicje podstawowych przyrostków, które opisałem w książce *SQL Programming Style*.

- Przyrostek **`_id`** oznacza identyfikator. Jest on unikatowy w schemacie i wszędzie tam, gdzie występuje w schemacie, określa jedną *encję*. W tabeli wyszukiwania występują atrybuty i ich wartości, a nie encje, dlatego z definicji takie identyfikatory nie występują w tego typu tabelach. To dlatego określenia takie jak **`_category_id`** czy **`_type_id`** to bardzo złe nazwy. Nigdy nie stosuj określenia `<nazwa_tabeli>`**`_id`**, które jest nazwą

opartą na lokalizacji i informuje, że kolumna prawdopodobnie nie zawiera prawdziwych kluczy. Samo `id` to zbyt niejasna nazwa, aby była dla kogokolwiek przydatna. Takie nazwy zanieczyszczają słownik danych, gdy trzeba sprawdzać mnóstwo takich samych nazw oznaczających różne elementy danych (często o identycznym, za dużym typie).

- Przyrostek **`_date`** (lub `dt`) oznacza wymiar czasowy. Dane oznaczają wtedy datę czegoś — zatrudnienia, urodzin, zwolnienia itd. Nie należy tworzyć kolumn o nazwie w postaci samego `date`.
- Przyrostek **`_nbr`** (lub `num`) oznacza numer identyfikacyjny. Jest to łańcuch cyfr (lub znaków alfanumerycznych) określający nazwę czegoś. Nie stosuj przyrostka **`_no`**, ponieważ wygląda on jak wartość logiczna (`yes` lub `no`). Ja chętniej stosuję `nbr` niż `num`, ponieważ jest to skrót często stosowany w kilku językach europejskich.
- Przyrostek **`_name`** lub `_nm` to nazwa składająca się z liter i nie wymaga objaśnień. Czasem zbiór takich nazw jest nazywany skalą nominalną.
- Przyrostek **`_code`** lub **`_cd`** oznacza standardowy kod opracowany przez zaufane źródło poza firmą. Na przykład kod pocztowy jest określany przez Pocztę Polską. Kod jest zrozumiały w określonym kontekście, dlatego nie zawsze trzeba go przekształcać, gdy jest wyświetlany ludziom.
- Przyrostek **`_size`** związany jest ze standardem branżowym lub firmowym i służy do określania wielkości towarów (na przykład ubrań, butów, kopert lub śrub). Zwykle istnieje definiujący rozmiary prototyp przechowywany przez zaufane źródło.
- Przyrostek **`_seq`** oznacza sekwencję (z numerami porządkowymi). Nie jest tym samym co numer identyfikacyjny, ponieważ nie może obejmować luk. Określone są też reguły dotyczące następników w sekwencji.
- Przyrostek **`_status`** określa wewnętrzne kodowanie opisujące stan, który może wynikać z wielu czynników. Na przykład kolumna **`credit_status`** może być obliczana na podstawie różnych źródeł. Słowo „status” pochodzi od słowa „state”. Można założyć, że istnieją określone dozwolone zmiany stanu. Na przykład stan cywilny może się zmienić na „rozwiedziony” tylko wtedy, gdy obecny stan to „żonaty”.
- Przyrostek **`_cat`** określa kategorię, czyli kodowanie oparte na zewnętrznym źródle z odrębnymi grupami encji. Do ustalania przynależności do kategorii powinny służyć bardzo wyraźne formalne kryteria. Przykładem jest klasyfikacja królestw w biologii.
- Przyrostek **`_class`** oznacza wewnętrzne kodowanie bez zewnętrznego źródła, opisujące podkategorię encji. Klasyfikacja powinna odbywać się na podstawie wyraźnych kryteriów formalnych. Przykładem jest tu klasyfikacja roślin w biologii.
- Przyrostek **`_type`** określa kodowanie, które wewnętrznie i zewnętrznie ma to samo znaczenie. Typy są zwykle mniej formalne od klas i mogą się pokrywać. Na przykład prawo jazdy można otrzymać na motor, samochód osobowy, taksówkę, ciężarówkę itd.

> Różnice między typami, klasami i kategoriami dotyczą rosnącej ścisłości algorytmu przypisującego encje do danego typu albo określonej klasy lub kategorii.

Kategorie są bardzo jednoznaczne. Rzadko trzeba zgadywać, czy coś jest zwierzęciem, warzywem, czy minerałem, by przypisać dany obiekt do jednej z tych kategorii.

Klasa to zestaw obiektów o pewnych cechach wspólnych. Istnieją reguły pozwalające stwierdzić, że zwierzę jest ssakiem lub gadem. W niektórych sytuacjach reguły nie są oczywiste (na przykład w przypadku składających jaja ssaków australijskich), przy czym wyjątki często są zaliczane do odrębnej kategorii (tu są nią stekowce).

Typy są najmniej ścisłe spośród trzech omawianych sposobów klasyfikowania i mogą wymagać subiektywnej oceny. Na przykład w niektórych stanach trójkołowce są uznawane za motocykle, natomiast w innych — za samochody. W jeszcze innych stanach trójkołowce są traktowane jak samochody tylko wtedy, jeśli mają bieg wsteczny.

W praktyce wszystkie trzy pojęcia często stosuje się w różnych kontekstach. Na przykład określanie wartości kolumny **`blood_type`** (z grupą krwi) odbywa się w wyniku procedury laboratoryjnej i ma prowadzić do jednoznacznego ustalenia wartości ze zbioru {A, B, AB, 0}. Zachowaj zgodność ze standardem branżowym, nawet jeśli narusza to przedstawione wcześniej definicje.

### 9.1.2. Błędy związane z typami danych i ograniczeniami

Pomyśl o używanym typie danych. Jeśli element danych jest używany w obliczeniach, musi być typu liczbowego. Nie powinien być łańcuchem znaków ani typem związanym z czasem. Podobnie dla danych reprezentujących czas stosuj typy związane z czasem, a nie łańcuchy znaków. Łańcuchy znaków są odpowiednie dla skal i numerów identyfikacyjnych. Szczegółowe informacje znajdziesz w książce *SQL Programming Style* (Morgan Kaufmann, 2005; ISBN: 978-0120887972).

Najczęstszy błąd dotyczący ograniczeń polega na ich pomijaniu. Ograniczenia są przekazywane do optymalizatora i poprawiają jakość kodu — chronią integralność danych. Po ich utworzeniu są używane dla każdej instrukcji wykonywanej na danym obiekcie zbudowanym za pomocą DDL-a. *Za każdym razem działają w ten sam sposób.*

## 9.2. Podział atrybutów

Podział atrybutów przyjmuje wiele form. Występuje, gdy używany jest jeden atrybut, ale jego wartości są umieszczane w schemacie w różnych miejscach. Wymaga to ponownego poskładania pierwotnego faktu, aby móc go zastosować. Najczęściej występującą formą podziału atrybutów jest tworzenie dla każdej wartości odrębnych tabel. Inna postać to tworzenie w tej samej tabeli odrębnych wierszy z częściami poszczególnych wartości. Te problemy łatwiej będzie zrozumieć na podstawie przykładów.

### 9.2.1. Podział atrybutów na poziomie schematu

Ten problem polega na umieszczeniu jednego schematu w wielu bazach. Zamiast tworzyć jedną bazę firmy, dane poszczególnych jednostek są zapisywane w odrębnych bazach. Podzielone bazy trzeba połączyć, by uzyskać kompletny model danych. Sytuacja ta różni się od podziału schematu na partycje. Schemat podzielony na partycje nadal jest jednostką logiczną.

### 9.2.2. Podział atrybutów na poziomie kolumny

Kolumna powinna być wartością *skalarną*. Nie oznacza to, że musi być wartością *atomową*. Określenie „atomowość" pochodzi od greckiego słowa „a-toma" oznaczającego „bez części". Skalarność oznacza, że dana wartość jest pomiarem na skali. Pomyśl o lokalizacji geograficznej. Para (długość, szerokość) określa *jedną* lokalizację, ale obejmuje *dwie* kolumny. W celu zlokalizowania obiektu na kuli ziemskiej możesz też zastosować jedną liczbę **HTM** (ang. *Hierarchical Triangular Mesh*).

Jeśli wartość nie jest atomowa, potrzebne są ograniczenia gwarantujące, że jej łączone komponenty są prawidłowe.

### 9.2.3. Podział atrybutów na poziomie tabeli

Gdybym zamierzał utworzyć bazę danych z osobnymi tabelami z męskimi i żeńskimi pracownikami, natychmiast zauważyłbyś, że należy zbudować tylko jedną tabelę z kolumną z kodem płci. Tabelę można potem podzielić na podstawie tej kolumny (`sex_code`).
Ta sytuacja jest oczywista, jednak zdarzają się też bardziej subtelne przypadki.

## 9.3. Problemy z przeciążaniem projektu

Niewłaściwy podział atrybutu skutkuje rozbiciem faktu na fragmenty w tabeli. Przeładowywanie to odwrotny problem — polega na umieszczaniu zbyt wielu elementów w schematach, tabelach i kolumnach.

### 9.3.1. Przeciążanie na poziomie schematu

Wymyśliłem zwrot „samochód, kalmar i Lady GaGa", aby opisać umieszczanie w jednostce logicznej pomieszanych i niepowiązanych danych. Na poziomie schematu oznacza to obciążanie jednej bazy danych zbyt wieloma zadaniami. Może tak się stać, gdy dwa systemy wyglądają podobnie, dlatego programista rozszerza istniejący model danych.

## 9.3.2. Przeciążanie na poziomie tabeli

Choć tabele można przeładować na wiele sposobów, koszmarem, który często się powtarza, są tabele **OTLT** (ang. *One True Look-up Table*). Związane jest to z umieszczeniem WSZYSTKICH kodowań w jednej dużej tabeli, gdy należy utworzyć osobne tabele dla poszczególnych kodowań. Myślę, że to Paul Keister był pierwszą osobą, która wymyśliła zwrot OTLT. Don Paterson (*http://www.sqlservercentral.com*) w jednym ze swoich artykułów nazwał tę technikę **MUCK** (ang. *Massively Unified Code-Key*).

Argumentem na rzecz takich tabel jest to, że potrzebna jest tylko jedna procedura do zarządzania *wszystkimi* kodowaniami i jedna generyczna funkcja do ich wywoływania (funkcja typu „samochody, kalmary i Lady GaGa"). Ta technika pojawia się nieustannie, warto jednak uhonorować Paula Keistera jako pierwszego, który nadał jej nazwę. Pomysł polega na tym, by utworzyć jedną tabelę do wyszukiwania wszystkich kodów ze schematu. Taka tabela zwykle wygląda mniej więcej tak:

```
CREATE TABLE OTLT -- Generyczne_Wyszukiwania?
(generic_code_type CHAR(10) NOT NULL, -- Okropne nazwy!
 generic_code_value VARCHAR(255) NOT NULL, -- Zwróć uwagę na wielkość!
 generic_description VARCHAR(255) NOT NULL, -- Zwróć uwagę na wielkość!
 PRIMARY KEY (generic_code_value, generic_code_type));
```

Elementy danych są teraz traktowane jak metadane, dlatego mają okropne nazwy. Kolumny nie reprezentują tu nic konkretnego — są „magicznymi" generycznymi miejscami na dowolne wartości. Dlatego jeśli schemat obejmuje znane z bibliotek kody **DDC** (ang. *Dewey Decimal Classification*), kody chorób w systemie **ICD** (ang. *International Classification of Diseases*) i dwuliterowe kody państw w standardzie ISO-3166, wszystkie te dane będą znajdować się w jednej wielkiej tabeli.

Zacznijmy od problemów z instrukcjami DDL, a następnie przejdźmy do okropnych kwerend, które trzeba pisać (lub ukrywać w widokach), gdy stosowana jest taka tabela. Wróćmy do pierwotnych instrukcji DDL i dodajmy ograniczenie `CHECK()` do kolumny `code_type`. W przeciwnym razie w wyniku literówki użytkownik może „wymyślić" nowy system kodowania.

Kody DDC i ICD są liczbowe i mają ten sam format. Składają się z trzech cyfr, kropki i dalszych cyfr (zwykle trzech). Jednak kody ISO-3166 są literowe. Ups — potrzebne będzie następne ograniczenie `CHECK`, które sprawdzi wartość kolumny `code_type` i ustali, czy łańcuch znaków ma właściwy format. Teraz tabela będzie wyglądać tak (jeśli oczywiście ktoś spróbuje zbudować ją w prawidłowy sposób, co zdarza się rzadko):

```
CREATE TABLE OTLT
(generic_code_type CHAR(10) NOT NULL
 CHECK(generic_code_type IN ('DDC', 'ICD', 'ISO3166', ..),
 generic_code_value VARCHAR(255) NOT NULL,
 CONSTRAINT Valid_Generic_Code_Type
```

```
CHECK (CASE WHEN generic_code_type = 'DDC'
            AND generic_code_value
            LIKE '[0-9][0-9][0-9].[0-9][ 0-9][ 0-9]'
            THEN 'T'
            WHEN generic_code_type = 'ICD'
            AND generic_code_value
            LIKE '[0-9][0-9][0-9].[0-9][0-9][0-9]'
            THEN 'T'
            WHEN generic_code_type = 'ISO3166'
            AND generic_code_value LIKE '[A-Z][A-Z]'
            THEN 'T' ELSE 'F' END = 'T'),
generic_description VARCHAR(255) NOT NULL,
PRIMARY KEY (generic_code_value, generic_code_type));
```

Ponieważ typowa baza danych w aplikacji może obejmować dziesiątki kodów, należy stosować się do przedstawionego wzorca tak długo, jak to konieczne. Efekt nie jest zachęcający, prawda? Pomyśl teraz, co się stanie po dodaniu nowych wierszy do tabeli OTLT.

```
INSERT INTO OTLT
(generic_code_type, generic_code_value, generic_description)
 VALUES ('ICD', 259.0, 'Inadequate Genitalia after Puberty'),
        ('DDC', 259.0, 'Christian Pastoral Practices & Religious
                        Orders');
```

Jeśli popełnisz błąd w kolumnie `generic_code_type` w trakcie wstawiania, aktualizowania lub usuwania danych, otrzymasz niewłaściwą wartość. Jeżeli w kwerendzie użyjesz błędnej wartości kolumny `generic_code_type`, efekty mogą być zaskakujące. Znalezienie takiego błędu może okazać się naprawdę trudne, gdy w jednym ze schematów o podobnych strukturach występują nieużywane kody.

Następną rzeczą, jaką warto zauważyć w tej tabeli, jest to, że używane są długie kolumny typu `VARCHAR(n)` lub, co jeszcze gorsze, typu `NVARCHAR(n)`. Ustawiana długość łańcuchów znaków (*n*) jest często jedną z największych dozwolonych w danym produkcie SQL-owym.

Ponieważ nie wiesz, jakie dane znajdą się w tabeli, nie da się przewidzieć i uwzględnić w projekcie bezpiecznej i rozsądnej wielkości maksymalnej. Ograniczenie wielkości trzeba umieścić w klauzuli `WHEN` w drugim ograniczeniu `CHECK()` dotyczącym kolumn `code_type` i `code_value`. Możesz też pogodzić się z istnieniem kodów o stałej długości większej niż konieczna.

Duża pojemność kolumny sprzyja wprowadzaniu błędnych danych. Jeśli udostępnisz kolumnę typu `VARCHAR(n)`, możesz otrzymać łańcuch znaków z wieloma spacjami i dziwnym znakiem na końcu. Jeżeli utworzysz kolumnę typu `NVARCHAR(255)`, ktoś może zapisać w niej buddyjską sutrę chińskimi znakami w kodowaniu Unicode.

Pomyśl teraz o problemach związanych z używaniem opisanej tabeli OTLT w kwerendach. Zawsze konieczne jest podanie wartości kolumny `generic_code_type`, a także szukanej wartości.

```
SELECT P1.ssn, P1.lastname, .., L1.generic_description
FROM OTLT AS L1, Personnel AS P1
WHERE L1.generic_code_type = 'ICD'
  AND L1.generic_code_value = P1.disease_code
  AND ..;
```

W tej przykładowej kwerendzie trzeba znać wartość kolumny `generic_code_type` powiązaną z kolumną `disease_code` z tabeli `Personnel`, a także wartości innych kolumn z kodami. Jeśli jednak wpiszesz błędną wartość kolumny `generic_code_type`, i tak możesz otrzymać jakiś wynik.

Ponadto trzeba uwzględnić dodatkowe zasoby na konwersję typów. Dla niektórych kodów bardziej naturalny może okazać się typ liczbowy niż `VARCHAR(n)`, aby zapewnić poprawne sortowanie danych. Dopełnianie łańcuchów cyfr zerami na początku wymaga dodatkowych zasobów i jest ryzykowne, jeśli programiści nie uzgodnili, ilu zer należy używać.

Gdy kwerenda jest wykonywana, trzeba pobrać całą tabelę wyszukiwania — nawet jeśli używanych jest tylko kilka kodów. Jeżeli jeden kod znajduje się na początku obszaru pamięci fizycznej, a drugi — na końcu, potrzebnych może być wiele operacji zmiany strony i zapisu danych w pamięci podręcznej. W trakcie aktualizowania tabeli OTLT trzeba zablokować ją do czasu wykonania zadania. Przypomina to noszenie ze sobą wielu tomów encyklopedii, gdy potrzebny jest tylko jeden artykuł z czasopisma.

Rozważ teraz dodatkowe koszty związane ze stosowaniem w tabeli dwuczęściowego klucza obcego:

```
CREATE TABLE EmployeeAbsences
(..
 generic_code_type CHAR(3) -- Niezbędna minimalna długość
 DEFAULT 'ICD' NOT NULL
 CHECK (generic_code_type = 'ICD'),
 generic_code_value CHAR(7) NOT NULL, -- Niezbędna minimalna długość
 FOREIGN KEY (generic_code_type, generic_code_value)
 REFERENCES OTLT (generic_code_type, generic_code_value),
 ..);
```

Teraz trzeba przekształcić typy znakowe, co oznacza dodatkowe koszty. Co gorsza, kody ICD mają naturalną wartość domyślną (`000.000` oznacza brak diagnozy), natomiast dla kodów DDC taka wartość nie istnieje. W starszych systemach kodowania często stosowano same dziewiątki, aby przedstawić wartość „różne”. Dzięki temu takie dane po sortowaniu trafiały na koniec raportów w platformach w COBOL-u. Podobnie jak nie istnieje „magiczny i uniwersalny” identyfikator, tak nie ma też „magicznej i uniwersalnej” wartości domyślnej. Opisane podejście oznacza więc utratę jednego z najważniejszych mechanizmów SQL-a.

Podejrzewam, że to rozwiązanie zostało wymyślone przez programistów obiektowych, którzy traktują je jak rodzaj polimorfizmu w SQL-u. Dla nich tabela jest klasą (choć jest to nieprawdą), dlatego powinna działać polimorficznie (choć tak się nie dzieje).

### 9.3.3. Przeciążenie na poziomie kolumn

Jeśli nazwa kolumny wymaga wielu członów, jest źle sformułowana. Przyjrzyj się tabeli reprezentującej arkusz z rejestrem przyjść i wyjść.

```
CREATE TABLE Clipboard
(emp_name CHAR(35) NOT NULL,
 signature_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 signature_type CHAR(3) DEFAULT 'IN' NOT NULL
 CHECK (signature_type IN ('IN', 'OUT')),
 PRIMARY KEY (emp_name, signature_type));
```

Rzeczywistą nazwą kolumny `signature_timestamp` powinno być `in_or_out_signature_timestamp`. Gdy chce się wykonać dowolną prostą kwerendę, trzeba podać dwa wiersze w złączeniu zewnętrznym typu *self-join*, aby otrzymać parę z czasem przyjścia i czasem wyjścia każdego pracownika.

Oto prawidłowy projekt:

```
CREATE TABLE Clipboard
(emp_name CHAR(35) NOT NULL,
 sign_in_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 sign_out_time TIMESTAMP, -- NULL oznacza bieżący czas
 PRIMARY KEY (emp_name, sign_in_time));
```

Jeden atrybut — czas pracy — trzeba zamodelować w standardowym SQL-u za pomocą *dwóch kolumn*. Tu jednak rozbito go na wiersze identyfikowane za pomocą kodu z metadanymi, informującego, który kraniec przedziału czasowego reprezentuje każda wartość. Gdyby wartościami były długość i szerokość geograficzna, natychmiast dostrzegłbyś problem i umieścił obie części jednego atrybutu (lokalizacji geograficznej) w tym samym wierszu.

W skrajnym przypadku w tym błędnym podejściu używana jest opcja z metadanymi do określania zawartości kolumny, która obejmuje zupełnie różne i niepowiązane atrybuty.

## 9.4. Nadmiarowość wynikająca z braku postaci normalnej

Przyjrzyj się tabeli używanej do obliczania odległości w linii prostej między podanymi na liście miastami.

```
CREATE TABLE City_Coords
(city_name VARCHAR(25) NOT NULL PRIMARY KEY,
 city_latitude FLOAT NOT NULL
 CHECK (city_latitude BETWEEN –1.57079633 AND + 1.57079633)
 city_longitude FLOAT NOT NULL,
 CHECK (city_longitude BETWEEN –3.14159265 AND +3.14159265));
```

Możesz użyć kwerendy ze wzorem na dystans między punktami na sferze, aby otrzymać odległość. Ta tabela jest bardzo zwięzła, jednak obliczenia matematyczne dla każdej pary miast generują pewne koszty.

Rozważ teraz tabelę o tym samym przeznaczeniu, która wyszukuje odległość między dowolnymi dwoma miastami z listy.

```
CREATE TABLE City_Distances
(origin_city_name VARCHAR(25) NOT NULL,
 destination_city_name VARCHAR(25) NOT NULL,
 PRIMARY KEY (origin_city_name, destination_city_name),
 CHECK (origin_city_name < destination_city_name)
 linear_distance DECIMAL (7,2) NOT NULL
 CHECK (linear_distance >= 0));
```

Teraz można posortować nazwy dwóch miast i sprawdzić odległość między nimi. Dla ($n$) miast tabela zawiera ~($n^2/2$) wierszy. Można stosować argumenty związane z wydajnością i innymi aspektami, ważne jest jednak to, że obie tabele wykonują w schemacie to samo zadanie. Obie są znormalizowane, *jednak jedna z nich jest nadmiarowa*. W branży informatycznej mówi się, że programista, który nosi dwa zegarki, nigdy nie wie, jaka jest godzina. Dlatego posługuję się zasadą „jeden fakt, jeden sposób, jedno miejsce, jeden czas" do definiowania normalizacji.

## 9.4.1. Sprzeczne relacje

Dalej opisana jest reguła biznesowa, która wydaje się sensowna, póki jej się bliżej nie przyjrzeć. Zespół sprzedawców odpowiada za każdego klienta, do którego przypisany jest jeden z członków danego zespołu (sprzedawca). Za pozostałych klientów zespół nie odpowiada. Dlatego dla danego klienta należy poszukać przypisanego do niego sprzedawcy, a gdy ten jest niedostępny, trzeba zgłosić się do zespołu. W takiej sytuacji występują dwie możliwości pozbycia się nadmiarowej relacji.

Nadmiarowość można wyeliminować, usuwając relację „jest odpowiedzialny za" między zespołem a klientem. Uzyskany model zawiera tyle samo informacji co przed wyeliminowaniem nadmiarowości.

```
-- To najprostsza tabela w schemacie
CREATE TABLE SalesTeams
(sales_team_id INTEGER NOT NULL PRIMARY KEY,
 sales_team_name CHAR(10) NOT NULL);

CREATE TABLE Salespersons
(sales_person_id INTEGER NOT NULL PRIMARY KEY,
 sales_person_name CHAR(15) NOT NULL,
 sales_team_id INTEGER NOT NULL
 REFERENCES SalesTeams(sales_team_id)
 ON UPDATE CASCADE);
```

Co zrobić, jeśli przypisany do klienta sprzedawca jest niedostępny? Potrzebna jest możliwość ustawienia wartości NULL i przejścia na poziom zespołu, by pomóc klientowi.

```
CREATE TABLE Customer_Assignments
(customer_id INTEGER NOT NULL PRIMARY KEY,
 responsible_sales_team_id INTEGER NOT NULL
 REFERENCES SalesPerson(sales_team_id)
 ON UPDATE CASCADE,
 responsible_sales_person_id INTEGER
 REFERENCES Salespersons(sales_person_id)
 ON UPDATE CASCADE);
```

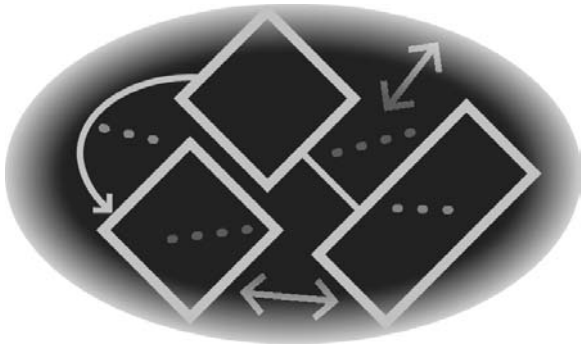
Druga możliwość polega na kontrolowaniu nadmiarowości za pomocą operacji DRI. Ponieważ dwa klucze obce wymagające synchronizacji znajdują się w tym samym wierszu, potrzebna jest tylko jedna aktualizacja. Zespół o danym **identyfikatorze** może (ale nie musi) obejmować sprzedawców, ale sprzedawca musi być przydzielony do jakiegoś zespołu.

```
-- To ta sama prosta tabela
CREATE TABLE SalesTeams
(sales_team_id INTEGER NOT NULL PRIMARY KEY,
 sales_team_name CHAR(10) NOT NULL);

CREATE TABLE Salespersons
(sales_person_id INTEGER NOT NULL PRIMARY KEY,
 sales_person_name CHAR(15) NOT NULL,
 sales_team_id INTEGER NOT NULL
 REFERENCES SalesTeams(sales_team_id)
 ON UPDATE CASCADE,
 UNIQUE (sales_person_id, sales_team_id));

CREATE TABLE Customer_Assignments
(customer_id INTEGER NOT NULL PRIMARY KEY,
 sales_team_id INTEGER NOT NULL,
 sales_person_id INTEGER
 REFERENCES Salespersons(sales_person_id)
 ON UPDATE CASCADE
 ON DELETE SET NULL,
 FOREIGN KEY (sales_person_id, sales_team_id)
 REFERENCES Salespersons (sales_person_id, sales_team_id)
 ON UPDATE CASCADE);
```


# Typy danych



# Skorowidz

## D



## E



## F



## G



## H



## I



## J

## K



## L



## Ł



## M



## N



## O

## P



## R



## S



## Ś



## T

## U



## W



## Z

# Poznaj SQL na poziomie eksperta!

SQL jest narzędziem stworzonym specjalnie na potrzeby baz danych. Nauka tego języka wymaga dogłębnego zrozumienia mechanizmów relacyjnych baz danych. Co więcej, w przypadku dużych zbiorów danych lub zbiorów danych wymagających szczególnego traktowania pozornie nieznaczne usterki kodu SQL mogą prowadzić do istotnych problemów w użytkowaniu aplikacji.

Niniejsza książka jest przeznaczona dla aktywnych programistów SQL. Przedstawia zaawansowane techniki i idiomy programistyczne z obszaru tego języka. Pokazano w niej również, w jaki sposób można rozwiązywać problemy, które często pojawiają się w bazach danych. Prezentowane tu rozwiązania można implementować właściwie we wszystkich systemach bazodanowych korzystających z SQL. To klasyczny podręcznik, systematycznie uaktualniany, który doczekał się prawdziwego uznania wśród specjalistów.

## W książce omówiono między innymi:

- transakcje i kontrolę współbieżności
- widoki oraz tabele i normalizację danych
- tabele pomocnicze, dane tymczasowe i operacje na danych, w tym na adresach IP, datach, danych tekstowych
- zaawansowane techniki tworzenia kwerend
- hierarchię w SQL, kolejki i macierze w SQL
- nowości w wydaniu: model z podwójnym zapisem czasu, funkcje agregujące, rozwiązywanie problemów z instrukcjami DDL i wiele innych zagadnień

## JOE CELKO

— uznany ekspert w dziedzinie baz danych. Z jego konsultacji korzystały m.in. służby medyczne, NASA, siły zbrojne USA. Autor wysoko cenionych książek dotyczących SQL. Napisał również ponad 1200 artykułów w prasie informatycznej i akademickiej, w większości poświęconych bazom danych.

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-2530-2
9 788328 325302
cena: 119,00 zł