Bazy Danych

2. DDL

Opracował: Maciej Penar

Spis treści

[1. Zanim zaczniemy 3](#_Toc510200720)

[2. Omówienie DDL 4](#_Toc510200721)

[Wstęp 4](#_Toc510200722)

[skrót wyrażeń 4](#_Toc510200723)

[Tworzenie tabel 5](#_Toc510200724)

[Usuwanie tabel 6](#_Toc510200725)

[Tworzenie prostych obiektów 7](#_Toc510200726)

[Tworzenie KOLEKCJI obiektów – poziom advanced 8](#_Toc510200727)

[dziedziczenie – poziom expert 9](#_Toc510200728)

[3. (3 pkt) Bonusowe punkty 12](#_Toc510200729)

[3. (12 pkt) Data Definition Language 13](#_Toc510200730)

[(8 pkt) Up.sql 13](#_Toc510200731)

[(2 pkt) down.sql 13](#_Toc510200732)

[(2 pkt) UPDATE.sql 13](#_Toc510200733)

# 1. Zanim zaczniemy

Zrelaksować się i przyswoić sobie teorię dot. SQL – w szczególności grup wyrażeń Data Manipulation Language (DML), Data Definition Language (DDL) oraz Data Control Language (DCL). Na chwilę odstawimy wyrażenia SELECT.

Materiały:

* SQL: <https://pl.wikipedia.org/wiki/SQL>
* Podstawowy kurs systemów baz danych, rozdział … o SQL’ach (nie mam książki przy sobie), J. Ullman, J. Widom

Oprogramowanie:

* ORACLE Database c12
  + SQL Developer

Fragmenty dokumentacji Oracle 12c:

* CREATE TABLE: [link](https://docs.oracle.com/database/121/SQLRF/statements_7002.htm)
* Indeksy: [link](https://docs.oracle.com/cd/B28359_01/server.111/b28310/indexes003.htm#ADMIN11722)
* Dziedziczenie: [link](https://docs.oracle.com/en/database/oracle/oracle-database/12.2/adobj/inheritance-in-sql-object-types.html#GUID-D6D92FB6-7BC4-4EE6-A9EC-BC69C5BA5A56)
* Przydatne funkcje dla obiektów: [link](https://docs.oracle.com/en/database/oracle/oracle-database/12.2/adobj/functions-and-operators-useful-with-objects.html#GUID-6CD85D07-1E06-4E42-A582-68F478040A39)
* Typy danych: [link](https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm)

# 2. Omówienie DDL

## Wstęp

Pozbiór SQL’a który nazywamy Data Definition Language (DDL) służy do modelowania bazy danych. Wyrażenia DDL na ogół nie zwracają danych, dlatego nazywa się je **poleceniami** (Commands), a nie **zapytaniami** (Queries).

Do najczęściej spotykanych wyrażeń DDL zaliczamy:

* **CREATE structure** – do tworzenia struktur w których przechowywane są dane np. CREATE TABLE
* **ALTER structure** – do zmiany istniejącej struktury np. ALTER TABLE
* **DROP structure** – do usunięcie istniejącej struktury np. DROP TABLE

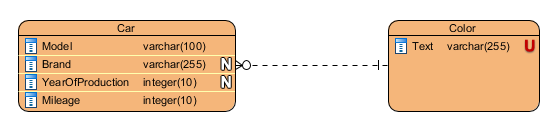
Dobrą praktyką w środowiskach produkcyjnych jest wykonanie poleceń utworzenia struktur (CREATE) jednokrotnie. Ewentualne zmiany przeprowadzane są z użyciem wyrażeń ALTER.

## skrót wyrażeń

* Dotyczące tabel:
  + CREATE TABLE
  + ALTER TABLE
  + DROP TABLE
* Dotyczące indeksów:
  + CREATE INDEX
  + DROP INDEX
* Dotyczące typów:
  + CREATE TYPE
  + DROP TYPE
* Dotyczące sekwencji:
  + CREATE SEQUENCE
  + DROP SEQUENCE
* Dotyczące widoków
  + CREATE VIEW
  + ALTER VIEW
  + DROP VIEW
  + CREATE MATERIALIZED VIEW
* Ograniczenia:
  + ADD CONSTRAINT
  + DROP CONSTRAINT
* Typy ograniczeń:
  + CHECK
  + FOREIGN KEY / REFERENCES
  + UNIQUE
  + PRIMARY KEY
  + NULL / NOT NULL
  + DEFAULT

## Tworzenie tabel

Załóżmy prostą bazę danych w której przechowujemy samochody, ich podstawowe informacje oraz kolory. Zdecydowaliśmy się zamodelować opis koloru samochodów jako osobną tabelę (tabelę słownikową). ERD wygląda następująco:



Z diagramu wynikają następujące ograniczenia:

* Marka oraz rok produkcji samochodu nie muszą być wpisane
* Model może mieć 100 znaków, Marka może mieć 255 znaków, Nazwa koloru może mieć 255 znaków
* Każdemu samochodu przypisany jest 1 kolor. W danym kolorze może istnieć wiele pojazdów.
* Nazwa koloru jest unikatowa w obrębie encji Color

Tworzenie baz danych zaczynamy od tworzenia tabel słownikowych. Tabelę Color tworzymy przez wyrażenie **CREATE TABLE:**

|  |
| --- |
| **CREATE TABLE** Colors(  ID **INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,**  TEXT **VARCHAR(255) UNIQUE NOT NULL**  ); |

Prześledźmy wyrażenie **CREATE TABLE** dla Colors**:**

* Linijka „**CREATE TABLE** [nazwa tabeli]” oznacza że chcemy utworzyć tabelę oraz umożliwia jej nazwanie
* Kolejne linie – wewnątrz nawiasów () – umożliwiają specyfikacje pól. W naszej tabeli są dwa pola:
  + Pole **ID** typu **INT,** które jest autogenerowane (**GENERATED ALWAYS AS IDENTITY**) I jest kluczem głównym (**PRIMARY KEY)**
  + Pole **Text** które jest typu **VARCHAR(255)** i jest unikatowe (**UNIQUE**) oraz niepuste (**NOT NULL**)

Utworzymy teraz tabelę dla samochodów:

|  |
| --- |
| **CREATE TABLE** Cars **(**  ID **INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,**  COLOR\_ID **INT REFERENCES** Colors(ID) **ON DELETE SET NULL,**  MODEL  **VARCHAR(100) NULL,**  BRAND  **VARCHAR(100) NOT NULL CHECK** (UPPER(BRAND) IN ('TOYOTA', 'BMW', 'LEXUS'))**,**  YEAR\_OF\_PRODUCTION **INT NULL CHECK** (YEAR\_OF\_PRODUCTION> 1900),  MILEAGE **INT DEFAULT 0,**  FRIENDLY\_NAME **AS** (Model || ' ' || Brand)  **);** |

Prześledźmy deklaracje pól w **CREATE TABLE** dla Cars**:**

* Pole **ID** typu **INT,** które jest autogenerowane (**GENERATED ALWAYS AS IDENTITY**) I jest kluczem głównym (**PRIMARY KEY)**
* Pole **COLOR\_ID** jest typu **INT** i wskazuje na pole ID w tabeli Colors. Gdy pewien kolor zostanie usunięty z tabeli Colors, pociągnie to aktualizację rekordów samochodów w tabeli Cars tak by wartości COLOR\_ID były równe NULL. Wyrażenie **ON DELETE SET NULL** jest opcjonalne, inne dostępne techniki w ramach tego wyrażenia to:
  + **ON UPDATE / ON DELETE –** precyzujące kiedy wymuszenie poprawności danych ma miejsce
  + **SET NULL / RESTRICT / CASCADE / NO ACTION –** precyzujące charakter podjętej akcji w ramach naruszenia więzów integralności (braku wskazywanego rekordu)
* Pole **MODEL** może mieć wartości NULL i maksymalną długość 100 znaków
* Pole **BRAND** nie może mieć wartości NULL i wartość musi należeć do zbioru ('TOYOTA', 'BMW', 'LEXUS')
* Pole **YEAR\_OF\_ PRODUCTION** może być wartością NULL, ale jak już ma wartość, to musi być większa od 1900
* Pole **MILEAGE** jest domyślnie inicjowanie wartością 0
* Pole **FRIENDLY\_NAME** którego typu nie znamy, ale jest wartością wyliczoną z kolumn **MODEL** i **BRAND** (znak || to konkatenacja)

Ogólnie rzecz biorąc wyrażenie **CREATE TABLE** ma następującą składnię:

|  |
| --- |
| **CREATE TABLE** [nazwa tabeli](  [nazwa pola 1] [typ pola 1] [ograniczenia pola 1],  [nazwa pola 2] [typ pola 2] [ograniczenia pola 2],  ….,  [nazwa pola N] [typ pola N] [ograniczenia pola N],  [dodatkowe ograniczenia]  ) [konfiguracja tabeli]; |

## Usuwanie tabel

Usuwanie tabel wykonujemy za pomocą polecenia „**DROP TABLE** [nazwa tabeli];”:

* **DROP TABLE** Cars;
* **DROP TABLE** Colors;

## Tworzenie prostych obiektów

Ciągniemy wątek samochody – kolory do granic absurdu. Załóżmy że mamy dobre powody żeby implementować naszą bazę danych jako relacyjno-obiektową bazę danych. Tworzenie obiektów zasadniczo nie różni się od tworzenia tabel.

Utworzenie klasy odbywa się za pomocą wyrażenia **CREATE TYPE** [nazwa typu] **AS OBJECT.**

Utworzymy klasę Color:

|  |
| --- |
| **CREATE TYPE** Color **AS OBJECT** (  TEXT **VARCHAR(255)**  ); |

Gdy mamy definicję klasy, możemy utworzyć na jej podstawie tabele relacyjno-obiektową za pomocą wyrażenia:

“CREATE TABLE [nazwa tabel] OF [nazwa typu];”

W naszym przypadku:

|  |
| --- |
| **CREATE TABLE** Colors **OF** Color; |

Dodajmy kilka obiektów do tabeli:

|  |
| --- |
| **INSERT INTO** Colors **VALUES**('Green');  **INSERT INTO** Colors **VALUES**('Red');  **INSERT INTO** Colors **VALUES**('Blue'); |

Dwa specjalne funkcje z których możemy korzystać w przypadku obiektów to:

* REF(x) – zwrócenie referencji do obiektu
* VALUE(x) – zwrócenie obiektu jako wartości

Dwa proste przykłady wykorzystania REF/VALUE:

|  |
| --- |
| **SELECT REF(c)** **FROM** Colors c;  **SELECT VALUE(c)** **FROM** Colors c; |

Teraz utwórzmy tabele Cars która będzie przechowywała nie klucz obcy do tabeli Cars, ale referencje na obiekty typu Car. Pogrubiam istotną linijkę.

|  |
| --- |
| CREATE TABLE Cars (  ID INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  **COLOR REF COLOR,**  MODEL VARCHAR(100) NULL,  BRAND VARCHAR(100) NOT NULL CHECK (UPPER(BRAND) IN ('TOYOTA', 'BMW', 'LEXUS')),  YEAR\_OF\_PRODUCTION INT NULL CHECK (YEAR\_OF\_PRODUCTION > 1900),  MILEAGE INT DEFAULT 0,  FRIENDLY\_NAME AS (Model || ' ' || Brand)  ); |

Debugowanie / walidacja relacyjno-obiektowych baz danych nie jest łatwa ☹. Dodawanie rekordów do tabeli Cars możemy przeprowadzić następująco:

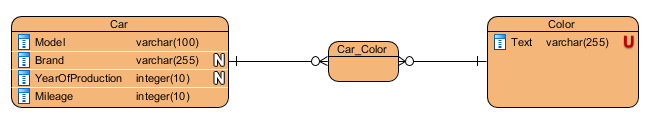
|  |
| --- |
| **INSERT INTO** Cars("COLOR", "MODEL", "BRAND", "YEAR\_OF\_PRODUCTION")  **VALUES***((SELECT REF(c) FROM COLORS c WHERE Text = 'Red')*,'Prius', 'Toyota', '2000'); |

Przykład odpytywania tabeli Cars:

|  |
| --- |
| **SELECT**  "ID",  "BRAND",  "MODEL",  **DEREF("COLOR").**"TEXT"  **FROM**  Cars; |

## Tworzenie KOLEKCJI obiektów – poziom advanced

Dobra… to jeszcze nic. Załóżmy że ktoś zauważył że ograniczenie: „Samochód ma 1 kolor”, jest za mocne. W takim scenariuszu stwierdzamy że rzeczywiste ERD powinno wyglądać:



Zdefiniujmy nowy typ który będzie kolekcją Referencji do Coloru (**REF COLOR**).

|  |
| --- |
| **CREATE TYPE** COLORSET **IS TABLE OF REF** COLOR; |

Typ **COLORSET** przechowuje kolekcję referencji typu COLOR.

Utworzenie tabeli Cars. Znowu pogrubiam linijkę istotną:

|  |
| --- |
| CREATE TABLE Cars (  ID INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  **COLORS COLORSET,**  MODEL VARCHAR(100) NULL,  BRAND VARCHAR(100) NOT NULL CHECK (UPPER(BRAND) IN ('TOYOTA', 'BMW', 'LEXUS')),  YEAR\_OF\_PRODUCTION INT NULL CHECK (YEAR\_OF\_PRODUCTION > 1900),  MILEAGE INT DEFAULT 0,  FRIENDLY\_NAME AS (Model || ' ' || Brand)  ) **NESTED TABLE** COLORS **STORE AS** NESTED\_CARS\_COLORS; |

Wstawianie rekordów do tabeli Cars:

|  |
| --- |
| **INSERT INTO** Cars("COLORS", "MODEL", "BRAND", "YEAR\_OF\_PRODUCTION")  **VALUES**(*(SELECT CAST(COLLECT(REF(c)) AS COLORSET) FROM COLORS c),*'Prius', 'Toyota', '2000'); |

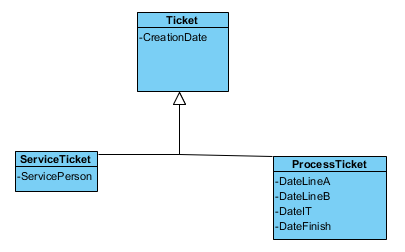
Odpytywanie tabeli zagnieżdżonej:

|  |
| --- |
| **SELECT**  c.BRAND,  c.MODEL,  DEREF(t.COLUMN\_VALUE).TEXT  **FROM**  CARS c,  TABLE(c.COLORS) t; |

## dziedziczenie – poziom expert

Żeby wyczerpać możliwości modelowania u klasycznych dostawców, zaprezentujemy tu jeszcze jedną metodę: dziedziczenie tabel/typów.

Załóżmy że chcemy zbierać dane przedstawione na następującym diagramie:



Mamy trzy byty:

* Ticket (zgłoszenie) – które ma swój czas wystąpienia
* ServiceTicket (zgłoszenie serwisowe) – które ma zarówno czas wystąpienia jak i osobę przypisaną do zgłoszenia
* ProcessTicket (zgłoszenie procesowe) – opisujące jakiś niezidentyfikowany proces – za pomocą stemplów czasowych

Utwórzmy typ bazowy **Ticket**:

|  |
| --- |
| **CREATE TYPE** TICKET **AS OBJECT**(  CREATE\_DATE TIMESTAMP  ) **NOT FINAL**; |

Istotne jest to żeby na końcu znalazło się ograniczenie **NOT FINAL –** wszystkie typy w Oracle są domyślnie oznaczane jako **final** (nie mogą być dziedziczone).

Utwórzmy typy pochodne **ServiceTicket**:

|  |
| --- |
| **CREATE TYPE** SERVICE\_TICKET **UNDER TICKET(**  SERVICE\_PERSON **VARCHAR(200)**  **);** |

Dziedziczenie odbywa się za pomocą wyrażenia UNDER [supertyp].

Na koniec stwórzmy typ pochodny **ProcessTicket:**

|  |
| --- |
| **CREATE TYPE** PROCESS\_TICKET **UNDER** **TICKET**(  DATE\_LINE\_A **TIMESTAMP**,  DATE\_LINE\_B **TIMESTAMP**,  DATE\_IT **TIMESTAMP**,  DATE\_FINISH **TIMESTAMP**  ); |

Utwórzmy tabelę TICKETS:

|  |
| --- |
| **CREATE TABLE** TICKETS **OF** TICKET; |

Hmm.. co możemy dodać? Zobaczmy. Na pewno typ TICKET:

|  |
| --- |
| **INSERT INTO** TICKETS **VALUES**(TICKET(SYSDATE)); |

Możemy też wsadzić podtypy… sorry za niewyszukany insert via PROCESS\_TICKET :P :

|  |
| --- |
| **INSERT INTO** TICKETS **VALUES**(SERVICE\_TICKET(SYSDATE, 'GB')); |
| **INSERT INTO** TICKETS **VALUES(**PROCESS\_TICKET(SYSDATE, SYSDATE, SYSDATE, SYSDATE, SYSDATE)); |

No dobra… jak to odpytać? Zapytanie:

|  |
| --- |
| **SELECT** \* **FROM** TICKETS t; |
| Zwraca:  CREATE\_DATE  ---------------------------  18/03/30 18:52:13,000000000  18/03/30 18:52:36,000000000  18/03/30 18:55:05,000000000 |

Hmm.. brakuje nam kolumn podtypów ☹

Może tak:

|  |
| --- |
| **SELECT** \* **FROM** TICKETS t **WHERE** VALUE(t) IS OF (SERVICE\_TICKET); |
| Zwraca:  CREATE\_DATE  ---------------------------  18/03/30 19:02:33,000000000 |

Dalej nie… Ale zapytanie:

|  |
| --- |
| **SELECT**  TREAT(VALUE(t) AS SERVICE\_TICKET).CREATE\_DATE AS CREATE\_DATE**,**  TREAT(VALUE(t) AS SERVICE\_TICKET).SERVICE\_PERSON AS SERVICE\_PERSON  **FROM**  TICKETS t; |
| Zwraca:  18/03/30 19:02:33,000000000 GB |

Nie jesteśmy poprawni. Wszystkie 3 obiekty posiadają CREATE\_DATE.

Zapytanie:

|  |
| --- |
| **SELECT**  CREATE\_DATE AS CREATE\_DATE,  TREAT(VALUE(t) AS SERVICE\_TICKET).SERVICE\_PERSON AS SERVICE\_PERSON  **FROM**  TICKETS t; |
| Daje poprawny wynik: |

Dziedziczenie umożliwia tworzenie takich zapytań:

|  |
| --- |
| **SELECT**  CREATE\_DATE AS CREATE\_DATE,  TREAT(VALUE(t) AS SERVICE\_TICKET).SERVICE\_PERSON AS SERVICE\_PERSON,  TREAT(VALUE(t) AS PROCESS\_TICKET).DATE\_LINE\_A AS DATE\_LINE\_A,  TREAT(VALUE(t) AS PROCESS\_TICKET).DATE\_LINE\_B AS DATE\_LINE\_B,  TREAT(VALUE(t) AS PROCESS\_TICKET).DATE\_IT AS DATE\_IT,  TREAT(VALUE(t) AS PROCESS\_TICKET).DATE\_FINISH AS DATE\_FINISH  **FROM**  TICKETS t; |
| Wynik: |

Ostatnią opcją jest możliwość ograniczenia się do konkretnego typu za pomocą wyrażenia

„**VALUE**( ) **IS OF**([typ])”

Przykład:

|  |
| --- |
| **SELECT**  CREATE\_DATE AS CREATE\_DATE,  TREAT(VALUE(t) AS SERVICE\_TICKET).SERVICE\_PERSON AS SERVICE\_PERSON  **FROM**  TICKETS t  **WHERE**  VALUE(t) IS OF(SERVICE\_TICKET); |
| Zwraca: |

Tyle.

# 3. (3 pkt) Bonusowe punkty

Na któryś zajęciach laboratoryjnych chciałbym pokazać dlaczego MS Access jest **bardzo** źle zaprojektowanym narzędziem. Dodatkowo na 3-ciej liście wracamy z SQL’em, ale do tego potrzebujemy porządnej Bazy Danych – można taką postawić podążając instrukcją <https://github.com/mpenarprz/BazyDanychI4/blob/master/Laboratorium/docx/1L.Oracle%20%26%20HammerDB.docx>

Z tego też względu przyznaję bonusowe punkty za:

* (1 pkt) za pokazanie że macie zainstalowanego lokalnie Access 2016
* (2 pkt) za pokazanie że macie zainstalowanego Oracle’a 12c i posadzoną bazę TPC-H

# 3. (12 pkt) Data Definition Language

Wybrać i opisać dowolną, sensowną mini-rzeczywistość – dająca się opisać w minimum 3 encjach. Określić jakie zapytania będą najczęściej wykonywane. W oparciu opisaną mini-rzeczywistość przygotować skrypty **Up.sql, Down.sql** oraz **Upgrade.sql**.

## (8 pkt) Up.sql

W tym skrypcie powinny znaleźć się wyrażenia **CREATE** które tworzą struktury oraz związki.

1. Napisać krótki opis mikro-świata
2. Utworzyć sekwencje przez **CREATE SEQUENCE**
3. Napisać wyrażenia **CREATE TABLE / CREATE TYPE**. Upewnić się że w każdej tabeli:
   1. Istnieje klucz główny, uzupełniany wartościami z sekwencji
   2. Dobrać odpowiednie typy danych
   3. Dobrane są odpowiednie ograniczenia (CHECK / NULL/ UNIQUE/DEFAULT)
   4. Zamodelowane są poprawne związki
   5. Nie ma wiszącej tabeli (takiej które nie jest wskazywana kluczem obcym z innej tabeli oraz nie posiada klucza obcego do innej tabeli)
4. Utworzyć widok eksponujący co najmniej dwie tabele
5. Określić jakie zapytania będą najczęściej wykonywane. Napisać polecenia **CREATE INDEX** pokrywające pola w zaproponowanych zapytaniach.
6. *\* Skrypt powinien być idempotentny*

## (2 pkt) down.sql

W tym skrypcie powinny znaleźć się wyrażenia **DROP** które zniszczą wszystkie obiekty.

1. Napisać polecenie **DROP** do wszystkich obiektów ze skryptu **Up.sql**
2. *\* Skrypt powinien być idempotentny*

## (2 pkt) UPDATE.sql

W tym skrypcie powinny znaleźć się wyrażenia **ALTER** które zmodyfikują istniejącą strukturę.

1. Napisać polecenie **ALTER TABLE,** które doda kolumnę wyliczalną (np. wykona uppercase/lowercase na kolumnie tekstowej, albo wyliczy hash ze wszystkich kolumn za pomocą funkcji STANDARD\_HASH [link](https://docs.oracle.com/database/121/SQLRF/functions183.htm#SQLRF55647))
2. Napisać polecenie **ALTER TABLE** które zmodyfikuje istniejącą kolumnę tekstową i zwiększy jej długość