T-SQL

język zapytań bazy danych MS SQL Server

Kurs dla początkujących

(C) 2008 Walter Łuszczyk

Spis treści

SPIS TREŚCI 2

PRZYGOTOWANIE ŚRODOWISKA TESTOWEGO 4

STRUKTURA BAZ DANYCH W ŚRODOWISKU MS SQL SERVER 2005 13

ZADANIA 18

LOGIKA RELACYJNYCH SYSTEMÓW BAZ DANYCH 20

ZADANIA 23

DROGA DO SUKCESU – MYŚLENIE W KATEGORIACH ZBIORÓW I INNE ZŁOTE RADY 24

POLECENIE SELECT 25

ZADANIA 32

SORTOWANIE WYNIKU ZAPYTANIA 34

ZADANIA 35

PRACA Z WARTOŚCIAMI TYPU NULL, FUNKCJE ISNULL ORAZ NULLIF 36

ZADANIA 37

ZŁĄCZENIA WEWNĘTRZNE I ZEWNĘTRZNE 38

ZADANIA 40

PODZAPYTANIA NIESKORELOWANE I SKORELOWANE 42

ZADANIA 44

TYPY DANYCH, KONWERSJE TYPÓW 45

ZADANIA 48

FUNKCJE WBUDOWANE 50

ZADANIA 54

GRUPOWANIE, FUNKCJE AGREGUJĄCE I FILTRY GRUP 56

ZADANIA 57

TABELE TYMCZASOWE I WYRAŻENIE WITH 59

ZADANIA 60

ZAUWAŻ, ŻE TYLKO RAZ MUSIAŁEŚ/AŚ DOKONYWAĆ UCIĄŻLIWEGO ŁĄCZRNIA TABEL.TROCHĘ PROGRAMOWANIA: DECLARE, SET, PRINT, IF ORAZ WHILE 60

TROCHĘ PROGRAMOWANIA: DECLARE, SET, PRINT, IF ORAZ WHILE 61

ZADANIA 62

KONSTRUKCJA CASE 63

ZADANIA 64

MODYFIKACJA DANYCH – POLECENIA: INSERT, UPDATE ORAZ DELETE 65

ZADANIA 66

TRANSAKCJE, CZYLI PROSTY SPOSÓB UNIKANIA POMYŁEK 68

ZAPYTANIA DYNAMICZNE I POLECENIE EXECUTE 69

CZEGO NIE ROBIĆ, CZYLI UŻYWANIE KURSORÓW I POLECENIA UNION 71

ZADANIA 72

OBRACANIE TABELI – OPERATOR PIVOT 73

Przygotowanie środowiska testowego

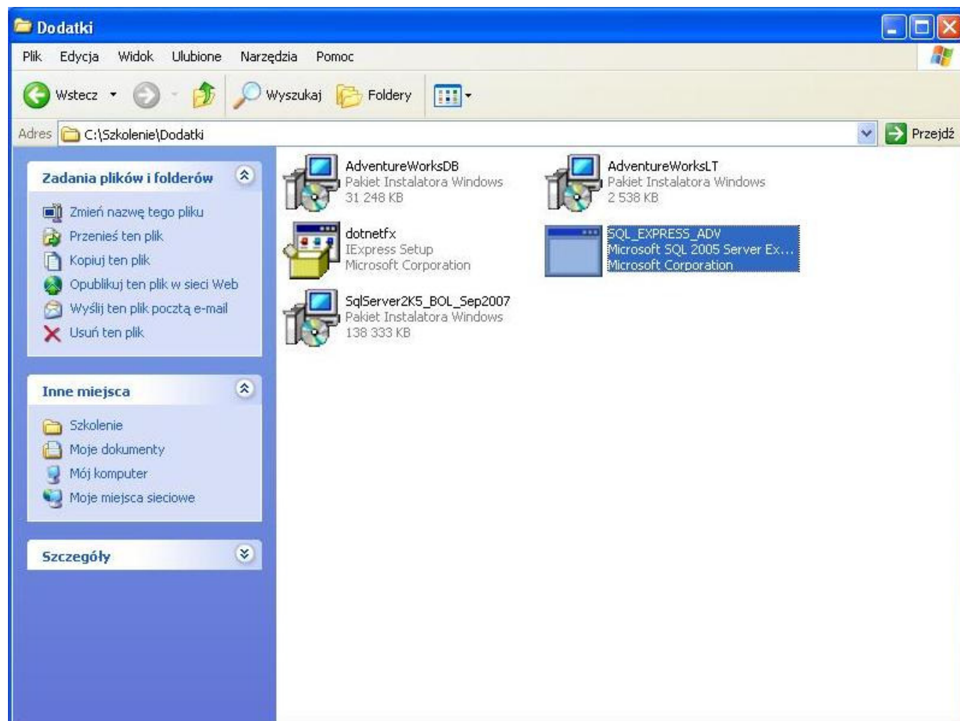
Do naszych ćwiczeń wykorzystamy darmową wersję *SQL Server-a Microsoft*, a mianowicie *MS SQL Server 2005 Express Edition*. Wersja ta ma pewne ograniczenia funkcjonalne, obsługuje maksymalnie jeden procesor, 1GB pamięci operacyjnej i może przechowywać bazy nie większe niż 1GB. Brak w niej jeszcze wielu zaawansowanych cech. Jednakże do naszych celów szkoleniowych baza ta w zupełności wystarczy. Zaletą tej wersji *SQL Server-a 2005* jest brak opłat za jej komercyjne używanie i rozpowszechnianie.

Wspomnianą edycję wraz z dodatkowymi komponentami można pobrać za darmo ze strony Microsoft-u: http://msdn2.microsoft.com/pl-pl/express/bb410792(en-us).aspx. Znajduje się tam też szczegółowa instrukcja jak zainstalować produkt, jednakże my prześledzimy dodatkowo ten proces.

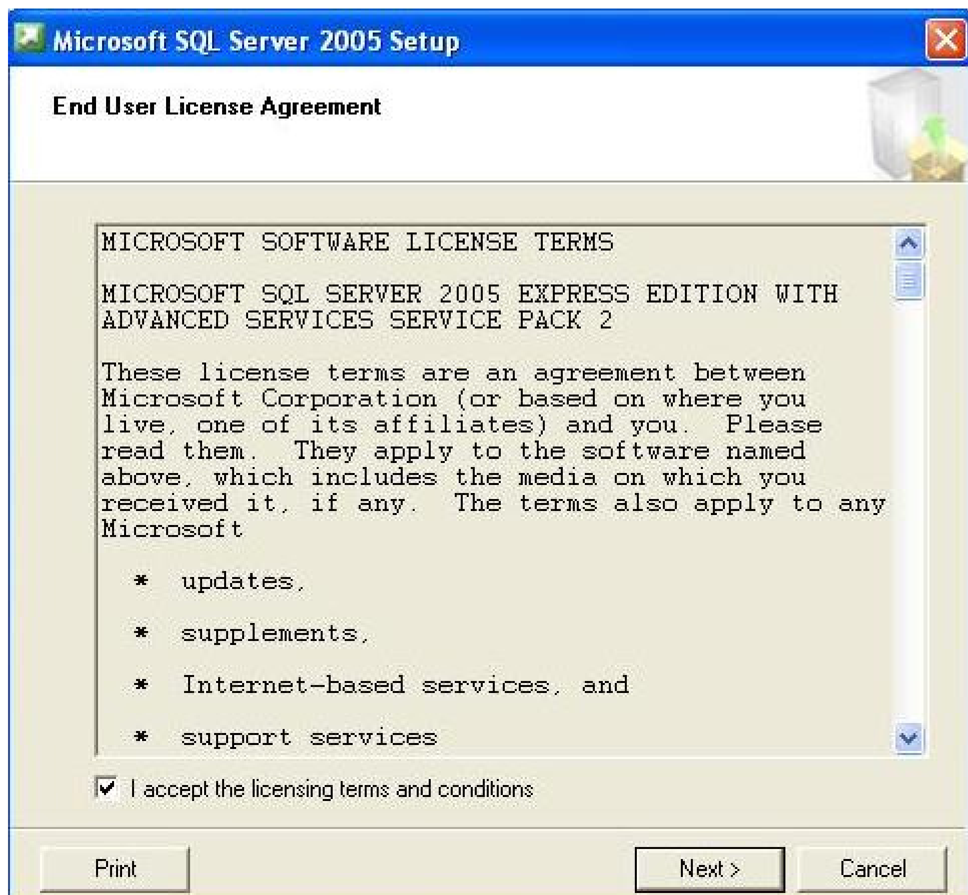
Uwaga! Jeśli na komputerze były zainstalowane wersje BETA oprogramowania *Framework 2.0* albo *SQL Server 2005*, to najpierw należy je odinstalować używając polecenia *Dodaj/Usuń programy* w *Panelu sterowania*.

Przez proces instalacji kompletnego środowiska ćwiczeniowego przeprowadzą nas następujące kroki:

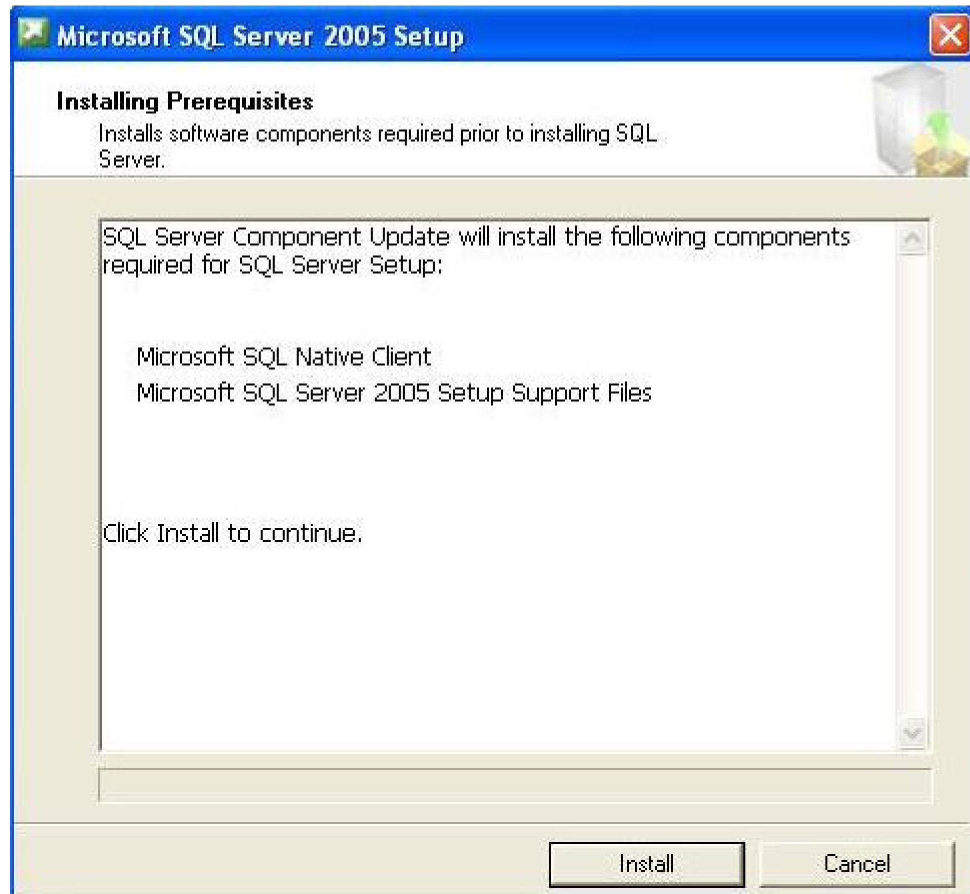
1. Otwórz zawartość głównego katalogu płyty CD dołączonej do szkolenia;
2. Jeśli mamy już zainstalowany dodatek *.NET Framework 2.0*, to punkt 2. należy pominąć. Aby rozpocząć instalację *.NET Framework-a 2.0* uruchom plik *dotnetfx.exe* znajdujący się w katalogu *Dodatki*; (można go także pobrać ze strony internetowej http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en);
3. Instalujemy *MS SQL Server 2005 Express* znajdujący się w pliku *SQL_EXPRESS_ADV.EXE* w katalogu *Dodatki* (można go także pobrać z podanej na samym początku strony). Przebieg instalacji:
    a. Uruchamiamy plik *SQL_EXPRESS_ADV.EXE*



b. W oknie *End User License Agreement* zanaczamy opcję *I accept the licensing terms and conditions* i klikamy klawisz *Next*



c. Na następnej stronie klikamy *Install*, aby zainstalować komponenty potrzebne do uruchomienia *SQL Server*-a;



Po zakończeniu tego procesu, klikamy *Next*;

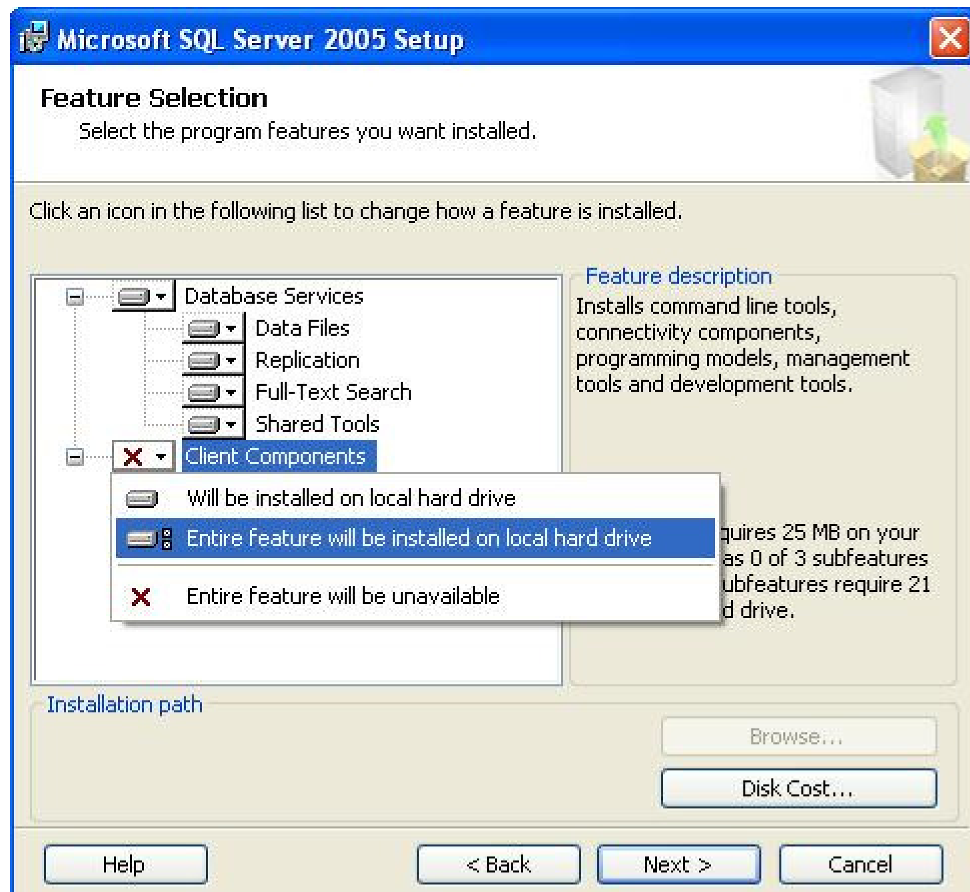
d. W oknie *Welcome to the Microsoft SQL Server Installation Wizard* klikamy *Next*;

e. Następuje teraz sprawdzenie konfiguracji komputera;

Klikamy *Next*;

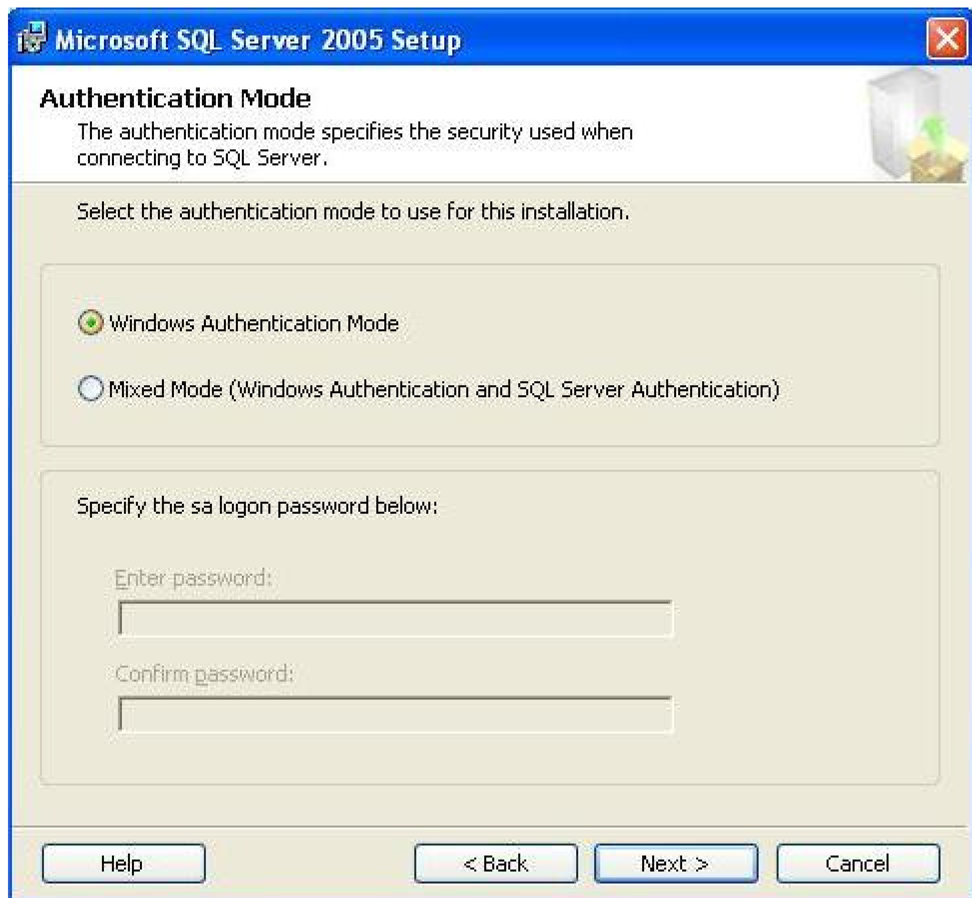
f. Wpisujemy swoje imię i nazwisko i klikamy *Next*;

g. W oknie *Feature Selection* klikamy prostokąty przy głównych węzłach (*Databases Services* i *Client Components*) i z menu wybieramy opcję *Entire feature will be installed on local hard drive.* Prostokąty powinny mieć białe tło i ikonę twardego dysku;



Klikamy *Next*;

h. Na stronie *Authentication mode* zostawiamy standardowo zaznaczoną opcję *Windows Authentication* i klikamy *Next* 3 razy, po czym klikamy *Install* i czekamy na zakończenie procesu instalacji.



i. Klikamy *Finish – SQL Server 2005 Express Edition* jest zainstalowany.

4. W dalszej kolejności zainstalujemy przykładową bazę danych. Aby to wykonać należy:

   a. Uruchomić plik *AdventureWorksLT.msi* (przykładowa baza jest też dostępna w internecie – patrz pierwszy link w tym rozdziale);

   b. Klikamy *Next*;

   c. W oknie License Agreement zaznaczamy opcję *I accept the terms in the license agreement* i klikamy *Next*;

   d. W następnym oknie klikamy *Next*, a potem *Install*;

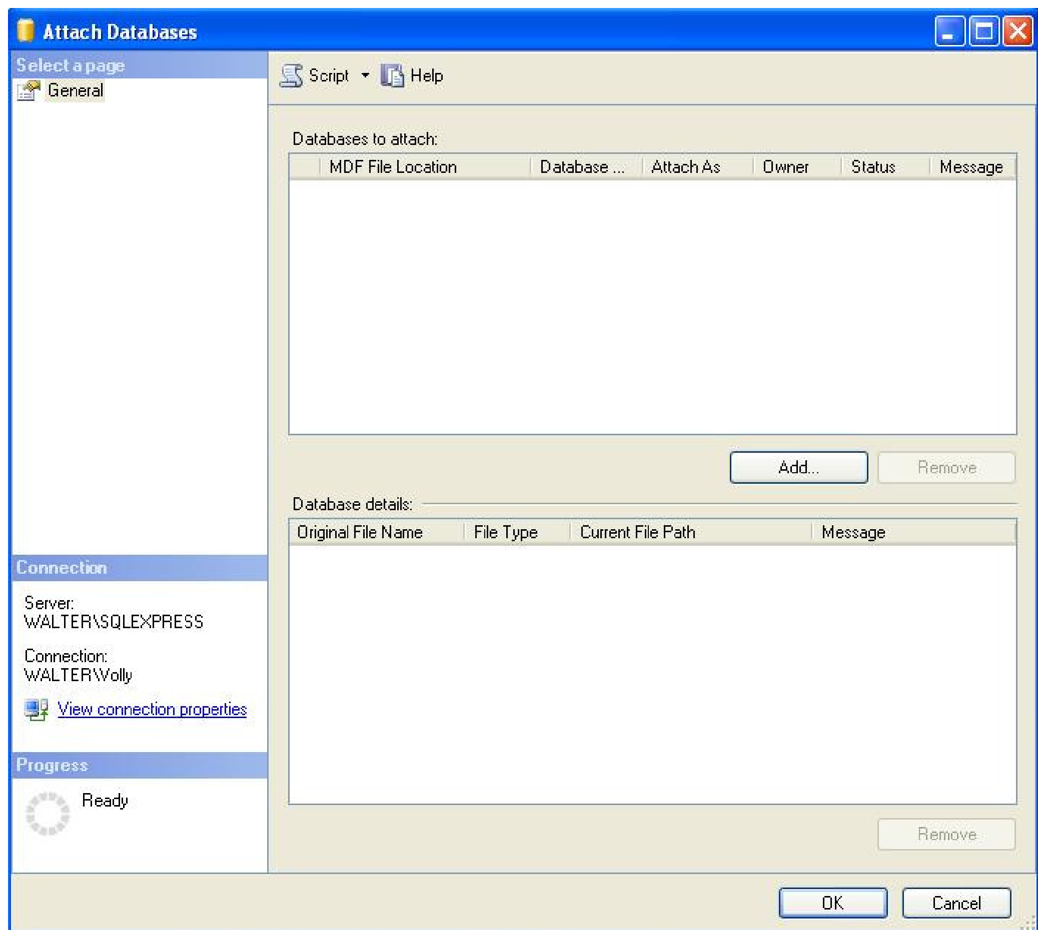
   e. Baza została zainstalowana, teraz należy ją podłączyć do naszego serwera;

   f. Uruchamiamy *SSMS*: *Start -> Wszystkie programy -> Microsoft SQL Server 2005 -> SQL Server Management Studio Express;*

g. W oknie *Connect to server* powinien wyświetlić się domyślny serwer, klikamy więc *Connect*;

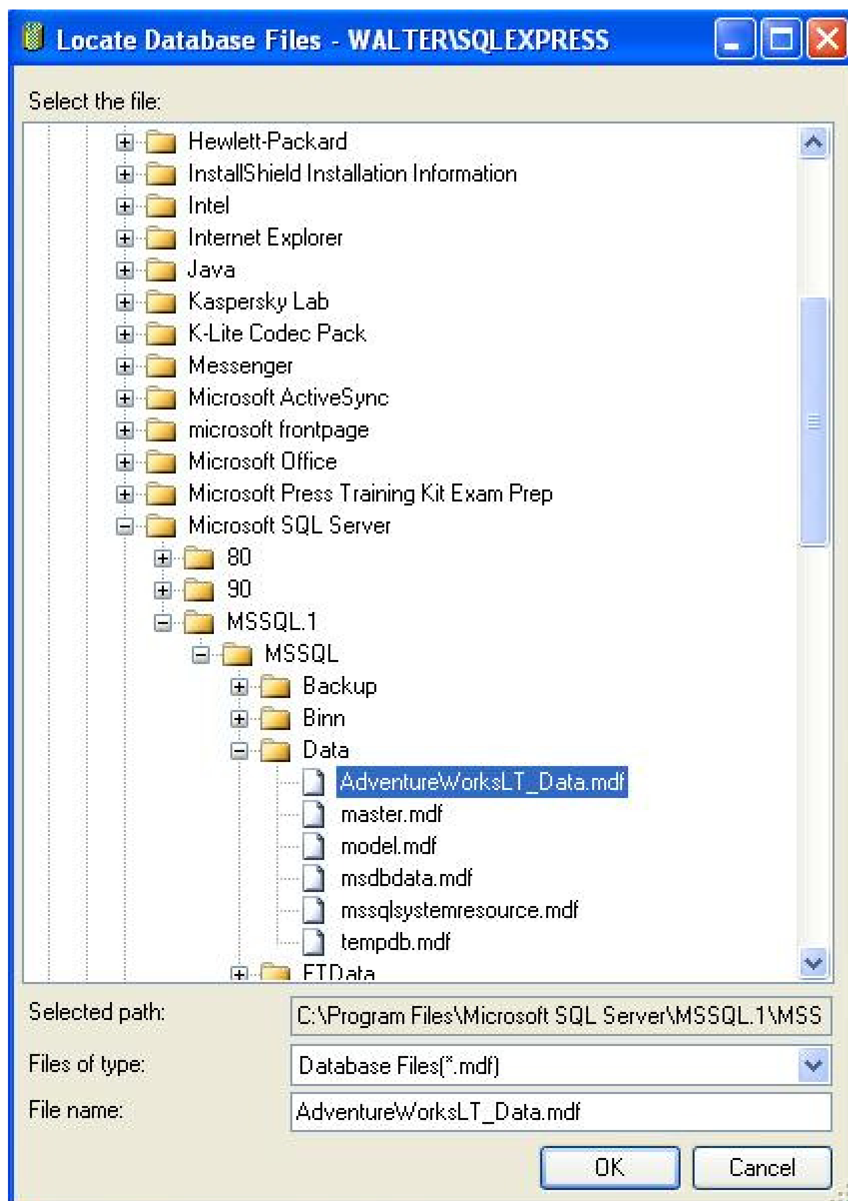
h. W oknie *Object Explorer* pojawił się dołączony serwer;

i. Klikamy prawym klawiszem myski na folderze *Databases* i z menu kontekstowego wybieramy *Attach*...



j. W oknie *Attach Database* klikamy *Add*;

k. W następnym oknie wybieramy plik *AdventureWorksLT_Data.mdf* i klikamy *OK*;



l. Klikamy *OK* w oknie *Attach Database*;

m. W *Object Explorer* rozwijamy folder *Databases*, aby upewnić się, że jest tam baza danych o nazwie *AdventureWorksLT*;

n. Zamykamy *SSMS*.

5. Ostatnim krokiem będzie zainstalowanie plików pomocy dla *SQL Server*-a (tzw. *SQL Books Online*):

   a. Uruchamiamy plik *SqlServer2K5_BOL_Sep2007.msi* znajdujący się w katalogu *Dodatki* (przykładowa baza jest też dostępna w internecie – patrz pierwszy link w tym rozdziale);

   b. Klikamy *Next*;

c. W oknie *License Agreement* wybieramy opcję I accept the terms in the license agreement i klikamy *Next*;
d. Podajemy swoje dane i klikamy *Next* 2 razy, a następnie *Install*;
e. Klikamy *Finish*;

W ten sposób środowisko jest przygotowane i gotowe do pracy.

Struktura baz danych w środowisku MS SQL Server 2005

Głównym narzędziem, którego będziemy używać podczas tego kursu to *SQL Server Management Studio* (*SSMS*). Jest to kompletne narzędzie pozwalające przeglądać strukturę bazy danych, wydawać zapytania do bazy, jak i prowadzić skomplikowane prace programistyczne.

Omówienie struktury zacznijmy od pojęcia serwera baz danych. Cytujemy za Helionicą, internetową encyklopedią:

Serwer baz danych - oprogramowanie komputerowe udostępniające innym komputerom usługi bazodanowe w modelu klient-serwer; także komputer, na którym takie oprogramowanie pracuje. Serwer baz danych zawiera oprogramowanie do obsługi baz danych i same bazy, na żądanie wyszukujące dane i udostępniające je klientom za pośrednictwem sieci.

Można powiedzieć, że serwer baz danych przechowuje bazy danych. I wydaje się, że ta definicja z naszego punktu widzenia jest najbardziej trafna. Wynika z niej, że kolejnymi interesującymi nas obiektami są bazy danych. **Baza danych** to zbiór danych zapisanych w ściśle określony sposób w strukturach odpowiadających założonemu modelowi danych. Serwer baz danych może przechowywać wiele baz danych. Głównym składnikiem baz danych są **tabele**, bo to one przechowują dane w odpowiedniej formie. Innymi interesującymi nas obiektami bazodanowymi są:

- **Widoki** (lub perspektywy) – są to wirtualne tabele, które mogą tworzyć inne „spojrzenie” na dane bez ich fizycznej reorganizacji. Znaczy to że możemy mieć widok pokazujący dane z dwóch innych tabel bez fizycznego przenoszenia tych danych. Przy wyciąganiu danych z bazy widoki są traktowane na równi z tabelami.

- **Procedury składowane** – kawałki kodu w języku T-SQL, który z reguły ma wykonać jakąś spójną logicznie operację; procedury mogą przyjmować parametry.

- **Funkcje** – jak wyżej, jednakże jest kilka szczegółów, którymi się różnią od procedur; funkcje możemy podzielić na systemowe (gotowe do użycia funkcje wbudowane w serwer, funkcje użytkownika zwracające pojedynczą wartość oraz funkcje użytkownika zwracające tabele).

Podczas tego kursu będziemy wykorzystywać powyższe obiekty, jednakże nie będziemy ich tworzyć. Więcej szczegółów na ich temat poznamy gdy zajdzie potrzba ich użycia.

Skupmy się teraz szczegółowiej na tabelach.

Tabela składa się z **kolumn** i **wierszy**. Tabelę definiuje zbiór kolumn i ich właściwości, tzn. typ i wielkość przechowywanych danych. Natomiast ilość danych zawartych w tabeli determinuje ilość wierszy. W tej samej kolumnie mogą być przechowywane dane tego samego typu, czyli np. liczby, napisy albo daty.

Innymi obiektami związanymi z tabelami oprócz zbioru kolumn są:

- **Klucze** – kolumny zawierające różne dane dla każdego wiersza; dzięki nim można uzyskać dostęp do konkretnego wiersza spośród wielu występujących;

- **Ograniczenia** – warunki zakładane na dane w kolumnach; możemy na przykład wymusić, że w kolumnie nie mogą być zapisane liczby większe niż 3000;
- **Wyzwalacze** (ang. triggers) – są to kawałki kodu w języku T-SQL (podobnie jak procedury) z tą różnicą, że wykonywane są tylko i wyłącznie po zmianie (wstawieniu, edycji lub usunięciu) danych w tabeli;
- **Indeksy** – specjalistyczne struktury danych, które umożliwiają szybkie wyszukanie interesujących nas danych spośród wielu tabel z wieloma milionami rekordów (wierszy);
- **Statystyki** – są to histogramy, które wraz z indeksami pozwalają na szybkie znajdowanie danych w bazie; są one z reguły tworzone i uaktualniane automatycznie;

Ćwiczenie 1. Przegląd obiektów przykładowej bazy danych *AdventureWorksLT.*

1. Uruchom *SSMS*;
2. W oknie *Connect to Server* kliknij *Connect*;

3. Zwróć uwagę, że w oknie obiektów (*Object Explorer*) pojawił się serwer o nazwie *<nazwa komputera>\SQLEXPRESS*; jeśli przyłączylibyśmy więcej serwerów, to pojawią się one w tym samym oknie poniżej;
4. Aby zobaczyć jakie bazy danych oferuje nam serwer, otwieramy folder *Databases*. Widzimy, że na serwerze jest baza danych *AdventureWorksLT* oraz podfolder *System Databases* (zawiera on własne bazy danych *SQL Server*-a potrzebne do działania). Otwieramy folder baz systemowych i stwierdzamy, że są to 4 bazy o nazwach *master*, *model*, *msdb* oraz *tempdb*;

5. Przyjrzyjmy się teraz dokładniej strukturze naszej przykładowej bazy *AdventureWorksLT*. Klikając znak plusa przy nazwie bazy *AdventureWorksLT* zaglądamy do jej struktury.

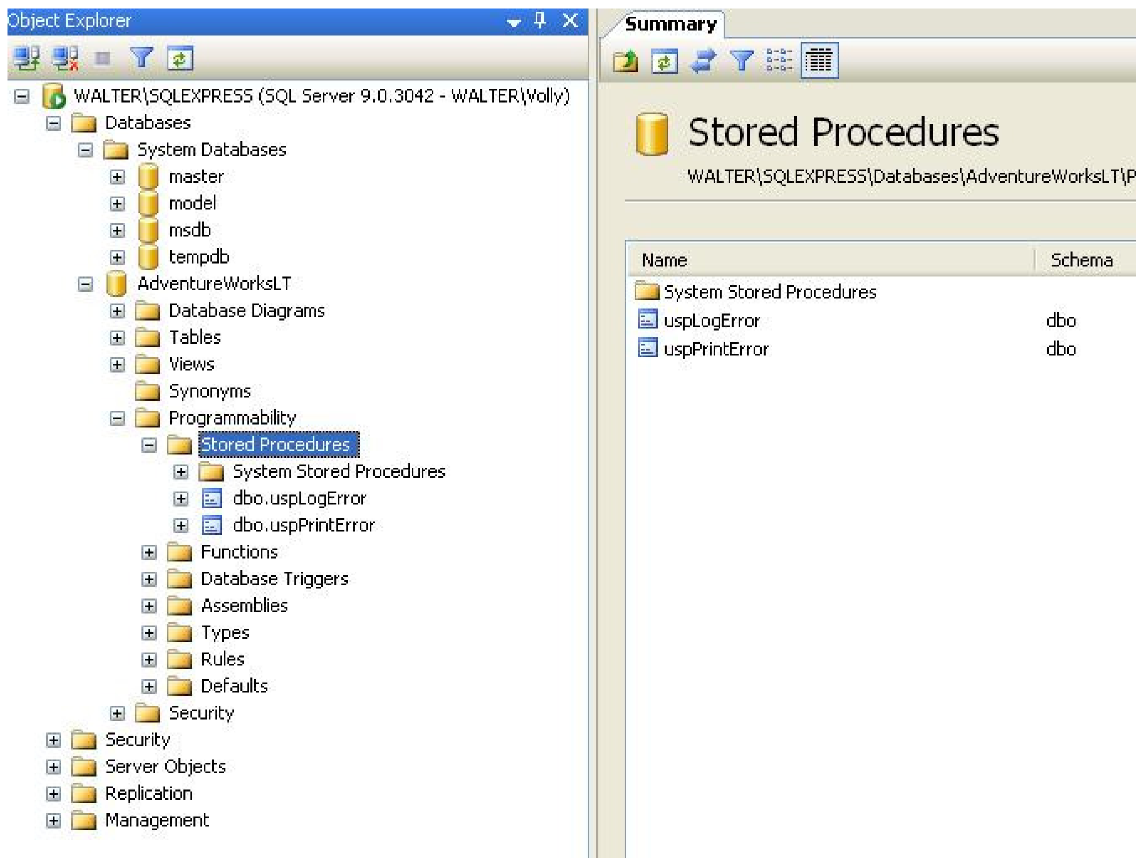
Znajdująsię tam foldery zawierające obiekty wymienione wcześniej jako składowe baz danych:

- ***Tables*** – zawiera wszystkie tabele bazy danych;
- ***Views*** – perspektywy;
- ***Programmability*** – funkcje i procedury;

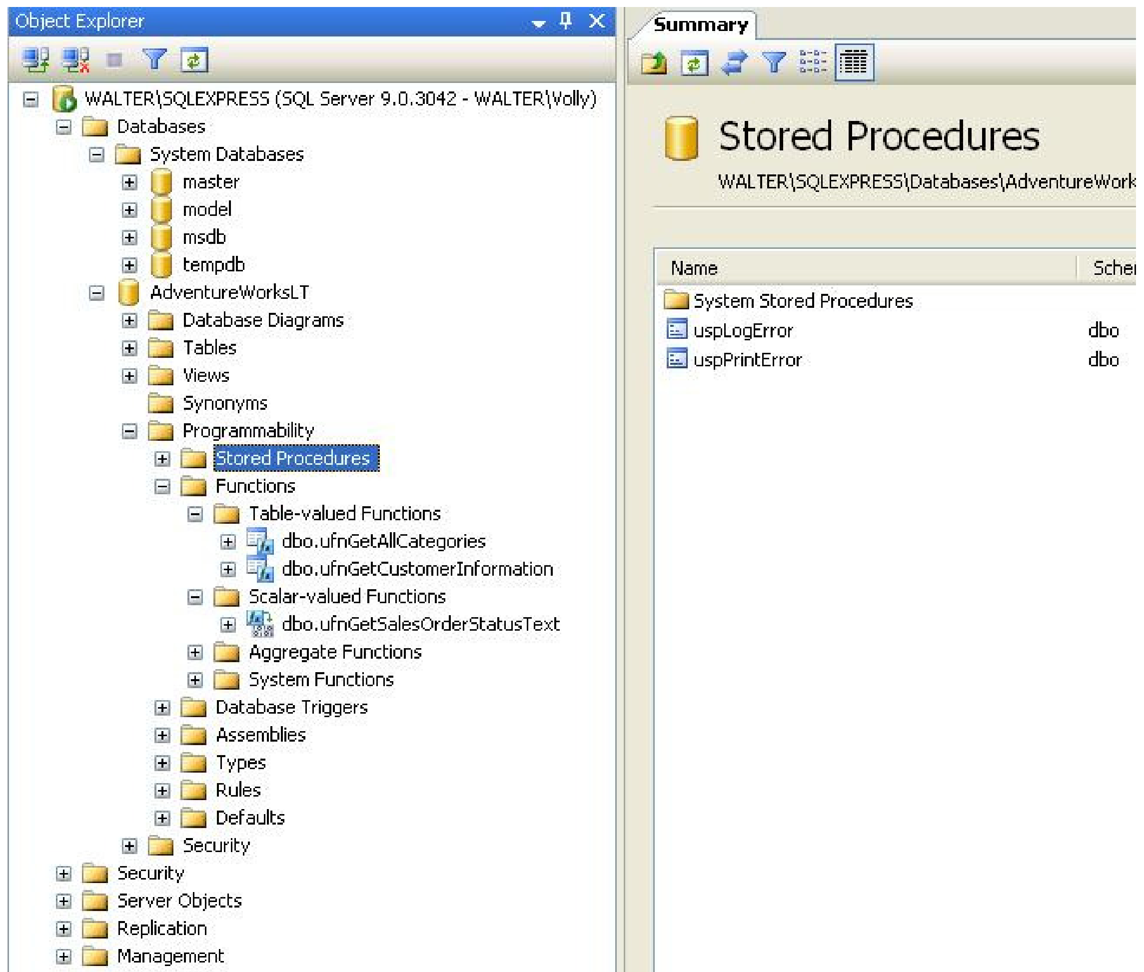
Są jeszcze inne foldery zawierające obiekty, o które są dla nas mniej istotne:

- ***Database Diagrams*** – umożliwia przechowywanie diagramów baz danych podobnych do tego dołączonego do skryptu; możliwe jest zobrazowanie zarówno całej bazy danych, jak i interesującego wycinka;
- ***Synonyms*** – synonimy obiektów bazy danych, dzięki ktorym jeden obiekt bazodanowy może mieć wiele nazw. Na przykład gdy mamy tabelę o nazwie Bailiff możemy nadać jej synonim Komornik, i po takiej nowej nazwie się do niej odwoływać;
- ***Security*** – folder zawierający elementy zabezpieczeń, jednakże tematyka ta wykracza poza ten kurs.

6. Wchodząc w głąb struktur bazo danowych, zacznijmy od procedur i funkcji. Aby wyświetlić listę procedur składowanych znajdujących się w bazie danych należy otworzyć folder *Programmability*, a następnie *Stored Procedures*.



Natomiast, aby zobaczyć listę dostępnych funkcji należy otworzyć folder *Functions* znajdujący się w folderze *Programmability*, a następnie *Table-valued Functions* (aby zobaczyć funkcje zwracające tabele) lub *Scalar-valued Functions* (aby zobaczyć funkcje zwracające pojedyncze wartości).



7. Zwińmy teraz folder *Programmability*, a rozwińmy folder *Tables* w celu przyjrzenia się tabelom i obiektom z nimi związanymi. Następnie znajdź tabelę *SalesLT.Product* i zajrzyj do środka klikając plusik po lewej stronie nazwy. Zauważmy, że są tam foldery przechowujące wszystkie elementy związane z tabelami, które wymieniliśmy w części teoretycznej tego rozdziału.

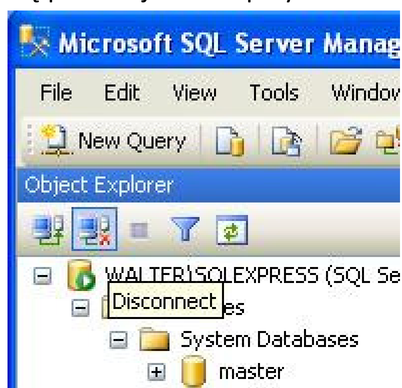
8. W przypadkach niespodziewanych zmian w bazie w momencie modyfikacji danych należy przejrzeć folder *Triggers*, aby upewnić się czy na tabeli nie mam jakiś wyzwalaczy, które wykonują dodatkowe operacje po naszej modyfikacji danych.

9. Innym interesującym nas folderem jest oczywiście foleder *Columns*, który zawiera kolumny definiujące tabelę. Rozwińmy ten folder aby zobaczyć z jakich kolumn składa się tabela *Product* oraz jakie są ich typy.

Ciekawą rzeczą, którą się można tu dowiedzieć jest informacja czy do kolumny można wstawiać wartość pustą (*NULL*). Jeśli w definicji jest napisane not null, to kolumna nie akceptuje wartości pustych, jeśli natomiast jest napisane *null*, to wartości takie są przyjmowane.

10. W ten oto sposób obejrzeliśmy strukturę bazy danych *AdventureWorksLT* za pomocą narzędzia *Microsoft SQL Server Management*;

11. Aby rozłączyć połączenie do serwera, z paska narzędzi okna *Object Explorer* wybierz *Disconnect*. Aby ponownie połączyć się z serwerem należy kliknąć ikonę *Connect*, znajdującą się po lewej stronie przycisku *Disconnect*.



Zadania

Jeśli będziemy pisać o bazie, to będziemy myśleć o bazie danych *AdventureWorksLT.*

1. Czy w bazie istnieje tabela o nazwie *Sales*?
2. Nazwy ilu tabel zaczynają się od słowa *Product*?
3. Wypisz wszystkie nazwy widoków, które znajdują się w bazie.
4. Czy w bazie istnieje procedura składowana o nazwie *dbo.uspPrintLog*?
5. Jakie parametry przyjmuje procedura *dbo.uspLogError* z bazy?
6. Jakie parametry przyjmuje procedura *dbo.uspPrintError* z bazy?
7. Z jakich kolumn składa się widok *vProductAndDescriptione*?
8. Czy w bazie, w tabeli *Product* istnieje kolumna o nazwie *SellEndDate*. Jeśli tak, to czy może się w niej znajdować napis „trzeci wrzesień”?
9. Czy w tabeli *Customer* istnieje kolumna *Salary*?
10. Czy w bazie istnieje funkcja *dbo.ufnGetAllCategories* zwracająca pojedynczą wartość?
11. Czy w bazie istnieje funkcja *dbo.ufnGetSalesOrderStatusText* zwracająca pojedynczą wartość?
12. Które funkcje z bazy zwracają tabele?

13. Czy dodając nowy adres do tabeli *Address* jest szansa na to, że w innej tabeli w bazie zostanie dokonana jakaś zmiana bez jawnego jej przeprowadzenia?
14. Czy dodając nowy adres do tabeli *SalesOrderDetail* jest szansa na to, że w innej tabeli w bazie zostanie dokonana jakaś zmiana bez jawnego jej przeprowadzenia?
15. Dodając nowy produkt do tabeli *Product* wystąpił błąd. Wewnątrz wiadomości znajduje się fragment: *CK_Product_SellEndDate*. Co może być przyczyną błędu?
16. Czy jest możliwe, że konstrukcja bazy przyspiesza wyszukiwanie adresów po kolumnie *StateProvince*?

Logika relacyjnych systemów baz danych

Jak już wspomnieliśmy w poprzednim rozdziale dane są przechowywane w tabelach. Dla każdej relacyjnej bazy danych można naryswoać schemat. Schemat naszej przykładowej bazy danych znajduje się na końcu skryptu. Na rysunku prostokąty oznaczają tebele, które przechowują zbiory obiektów, a linie pomiędzy nimi obrazują relacje. Weźmy dla przykładu relację pomiędzy tabelami *Product* i *ProductCategory*. Mimo, że na rysunku relacje nie mają etykiet, intuicja podpowiada, że każdy produkt jest w jakiejś kategorii, czyli śmiało można byłoby dodać etykietę „jest w”. Zagadką jeszcze przez chwilę pozostanie kierunek strzałki. Wyjaśnimy to nieco później.

Wróćmy na chwilę do tabel. W nagłówku prostokąta przedstawiającego tabelę jest jej nazwa. Zaraz poniżej do pierwszej kreski są kolumny będące kluczem głównym (*ang. primary key*) tabeli. Oznacza to, że za pomocą tych kolumn (lub często jednej kolumny) możemy wybrać dokałdnie jeden interesujący nas element ze zbioru, który jest zawarty w tabeli. Bardzo często jako kluczy używa się kolejnych liczb naturalnych nazywanych identyfikatorem. W przypadkach takich nazwa kolumny przechowującj te klucze ma następującą nazwę: *<nazwa tabeli>ID*, na przykład *ProductID*. Dzięki kluczowi możemy bardzo łatwo wskazać dowolny produkt ze zbioru produktów znajdującego się w tabeli *Product* przez określenie, że chcemy produkt o *ID = 456*. Jako, że kolumna *ProductID* jest kluczem, to mamy gwarancję, że taki produkt jest tylko jeden. Poniżej kluczy znajdują się kolumny zawierające dodatkowe atrybuty obiektów przechowywanych w tabeli. Przykładowo, jeśli tabela przechowuje zbiór adresów, to ich dodatkowymi atrybutami może być nazwa miasta, kod pocztowy, kraj, itp.

Przyjrzyjmy się teraz nieco głębiej relacjom. Wyróżniamy kilka rodzajów relacji, a o ich rodzaju na schemacie bazy mówią strzałki. Oto rodzaje możliwych relacji:

Relacja **„jeden do jedengo”** to relacja, w która dla jednego wiersza z jednej tabeli przyporządkowywuje jeden wiersz z innej tabeli. Przykład:

OsobaID	Imię	Nazwisko
1	Piort	Iksiński
2	Ksawery	Abacki
3	Jan	Rychter
4	Stanisław	Turski
5	Konstanty	Nowak

OsobaID	RokUrodzenia
1	1960
2	2007
3	1980
4	2000
5	1990

Obie tabele przedstawiają dane osób, które zostały ponumerowanie unikalnymi identyfikatorami według systemu opisanego wyżej. Między takimi tabelami może istnieć relacja jeden do jednego, ponieważ jednemu imieniu i nazwisku odpowiada dokładnie jedna data urodzenia. Relacje tego typu stosuje się dość rzadko, ponieważ najczęściej wszystkie atrybuty danego obiektu trzymamy w jednej tabeli. Można użyć tej relacji, aby rozbić jedną wielką tabelę na kilka mniejszych osiągając w ten sposób lepszą wydajność. Relacji tej nie ma na naszym schemacie. Oznacza się ją linią zakończoną strzałkami po obu stronach.

Następnym typem jest relacja **„jeden do wielu”** i jest to najczęstszy przypadek. Relacja ta ma miejsce kiedy chcemy jednemu obiektowi przyporządkować wiele innych, na przykład:

- W jednym dziale pracuje wielu pracowników;
- Wiele banknotów znajduje się w jednym portfelu;
- W jednym rządzie jest wiele ministrów;

Można byłoby tu tak na prawdę wyróżnić dwa rodzaje relacji, mianowicie „jeden do wielu" i „wiele do jednego" jednakże jest to relacja symetryczna i pierwszą z drugiej łatwo uzyskać zamieniając miejscami tabele. Relację taką realizuje się w następujący sposób. Wskazujemy tabelę która obpowiada za pojedyńczy obiekt w relacji i ustalamy w niej kolumnę zawierającą klucz główny (najczęściej jest to unikalny ID). W drugiej z tabel, reprezentującej obiekty, których jest więcej, dodajemy kolumnę, która dla każdego wiersza będzie zawierać klucz główny obiektu nadrzędnego. Spójrzmy na przykładową relację „w jednym dziale pracuje wielu pracowników".

PracownikID	DzialID	Imię	Nazwisko
1	1	Piort	Iksiński
2	2	Ksawery	Abacki
3	1	Jan	Nowak
4	1	Stanisław	Turski
5	3	Konstanty	Nowak

DzialID	Nazwa
1	Dział Produkcji
2	Dział Kontroli
3	Dział Kontaktów

Na schemacie relacje tego typu oznacza się linią zakończoną strzałką skierowaną do tabeli zawierającej obiekt występujący w relacji jednostkowo.

Ostatnim omawianym typem relacji jest **„wiele do wielu"**, czyli takiej relacji w której dla jedengo obiektu z lewej strony może odpowiadać wiele obiektów z prawej strony, jak i odwrotnie, czyli dla jednego obiektu z prawej strony może istnieć wiele przyporządkowanych obiektów z lewej strony. Przykładami takich relacji są:

- Jedno zamówienie może zawierać wiele rodzajów produktów, ale rózwnież produkt jednego rodzaju może występować w wielu zamówieniach;
- Jedeno ciasto składa się z wielu składników; jeden składnik występuje w wielu ciastach.

Aby zrealizować relację **„wiele do wielu"** potrzebna jest nam dodatkowa tabela oraz użycie dwóch relacji „jeden do wielu". Czasami jest też więcej atrybutów relacji. Jako przykład rozważmy tu relację pomiędzy osobami i adresami. Pod jednym adresem może mieszkać wiele osób, ale też jedna osoba może mieć wiele adresów (np. zameldowania, koresponedencyjny, itp). Oto tabele (*Osoba*, *Adres* i *OsobaAdres*) realizujące tą relację:

OsobaID	Imię	Nazwisko
1	Piort	Nowak
2	Ksawery	Abacki
3	Jan	Turski
4	Stanisław	Turski
5	Konstanty	Nowak

AdresID	Ulica	KodPocztowy	Miasto
1	Legnicka 50	55-555	Wrocław
2	Batorego 1	02-222	Warszawa
3	Neptuna 456	34-444	Gdańsk

OsobaAdresID	OsobaID	AdresID	Korespondencyjny
1	3	2	TAK
2	3	1	NIE
3	4	2	TAK
4	5	3	TAK
5	1	3	TAK

Można powiedzieć, że istnieje jeszcze jeden typ relacji, choć realizowany jest on za pomocą relacji jeden do wielu. Jednakże sposób użycia tej relacji jest szczególny, bo relacja jest skierowana od danej tabeli do niej samej. Relacje tego typu to **hierarchie**. Przykładami hierarchii są:

- „jest w kategorii" – kategorie można dowolnie zagnieżdżać, tzn. kategoria A może być w kategorii B, a ta z koleii w kategorii wyższego rzędu C, itd.;
- „jest szefem" – analogicznie jak powyżej.

Dzięki powyższej strukturze możliwe jest zagnieżdżanie dowolnej ilości obiektów wewnątrz obiektów tego samego typu na dowolną głębokość. Pozwala na przechowywanie w wygodny sposób struktur drzewiastych. Spójrzmy na realizację hierarchii „jest szefem":

PracownikID	SzefID	Imię	Nazwisko
1	2	Piort	Nowak
2	4	Ksawery	Abacki
3	4	Jan	Turski
4	NULL	Stanisław	Turski
5	2	Konstanty	Nowak

Została tu dodana kolumna *SzefID*, która zawiera tą samą logicznie wartość co kolumna *PracownikID*, czyli unikalny identyfikator (ID) pracownika, który też oczywiście może być szefem. W kolumnie *PracownikID* numeruje się kolejne obiekty, tak jak było to poprzednio, natomiast w kolumnie *SzefID* zawarty jest numer pracownika, który jest szefem pracownika z danego wiersza. I tak na przykład *Konstanty Nowak* ma szefa o *ID = 2*, czyli *Ksawerego Abackiego*, który jest z kolei podwładnym *Stanisława Turskiego*. Ten z kolei nie ma już nikogo nad sobą, co oznaczamy wartością pustą (*NULL*) w kolumnie *SzefID*.

Istota relacji zostanie uwypuklona przy zapytaniach wyciągających dane z wielu tabel. Wtedy relacje okażą się wskazówką jak połączyć dane z wielu tabel w sposób logiczny.

Zadania

1. Znajdź na załączonym do skryptu schemacie wszystkie relacje następujących typów:
   a. Jeden do jednego;
   b. Jeden do wielu;
   c. Wiele do wielu;
   d. Hierarchia;
2. Opisz właściwymi etykietami linie przedstawiające relacje na schemacie.
3. Jakie kolumny wchodzą w skład klucza głównego tabeli *ProductModelProductDescription*?
4. Jakie są klucze obce w tabeli *Product* i jakie kolumny je realizują?
5. Czy w tabeli *Product* istnieje kolumna wyliczana? Jeśli tak, to jaką ma nazwę?
6. Czy na kolumnach tabeli *Address* założone są indeksy? Jeśli tak, to ile ich jest i jakie kolumny wchodzą w ich skład?
7. Podaj po 5 przykładów następujących typów relacji:
   a. Jeden do jednego;
   b. Jeden do wielu;
   c. Wiele do wielu;
   d. Hierarchia;

Droga do sukcesu – myślenie w kategoriach zbiorów i inne złote rady

Omówimy sobie w tym rozdziale kilka kluczowych kwestii mówiących o tym jak lekko i łatwo przebrnąć przez pisanie zapytań w języku SQL.

Pierwsza rada – myśl w kategoriach zbiorów. Tworząc zapytanie należy pamiętać, że tabele przechowują zbiory elementów, a nie tylko „jakieś" wiersze. Kluczem do tworzenia dobrze działających zapytań języka T-SQL jest myślenie w kategoriach zbioru, a nie w kategoriach operacji wiersz po wierszu.

Druga rada – sformułuj najpierw pytanie w języku polskim opisujące to, co chcesz wyciągnąć z bazy danych.

Trzecia rada – konstruuj zapytania zaczynając od prostego zalążka, rozwijając go aż do uzyskania zapytania zwracającego żądany wynik.

Czwart rada - jest wiele sposobów na jeden wynik. Jeśli czujemy, że nasze zapytanie jest bardziej złożone, niż być powinno, to prawdopodobnie tak włąłaśnie jest. Cofnijmy się o krok i przemyślmy problem ponownie.

Rada piąta – nie kompliku zbytnio zapytania dla zachowania jego którtkiej treści. Prawdą jest, że pisanie zwięzłych i efektywnych zapytań jest sztuką, do której dążymy. Jednakże czasem warto aby zapytanie było dwa razy obszerniejsze, ale żebyśmy po 2 miesiącach jeszcze wiedzieli co ono robi i jak to robi.

Rada szósta – opisuj za pomocą komentarza każde bardziej złożone zapytanie. Najlepiej napisać w języku naturalnym jakie dane wyciąga z bazy. Mamy wtedy pewność, że gdy wrócimy do zapytania po dłuższej przerwie w celu jego modyfikacji, to zaoszczędzimy sporo czasu, a napewno więcej niż zajęło nam jego opisanie.

Rada siódma – Ćwiczyć! Ćwiczyć! I jeszcze raz ćwiczyć!

Polecenie SELECT

W poprzednich rozdziałach poznaliśmy na tyle bazy danych, że możemy już zacząć pobierać z nich dane. Polecenie *SELECT* (*pol. Wybierz*) jest podstawową komendą wydawaną *SQL Serwer*-owi. Jego prosta postać ma następującą składnię:

SELECT <nazwy kolumn rozdzielone przecinkami lub znak „”>*
FROM <nazwa tabeli>
WHERE <warunki na wybierane rekordy (wiersze)>

W pierwszej lini polecenia określamy kolumny, które chcemy obejrzeć. Nazwy kolumn możemy zastąpiś symbolem gwiazdki (*) jeśli chcemy zobaczyć dane ze wszystkich kolumn. Następnie w klauzuli *FROM* wpisujemy nazwę tabeli, z której będziemy wyciągnąć dane. Ostatni wiersz *WHERE* jest opcjonalny i umożliwia ograniczenie ilości zwracanych rekordów, do tych które spełaniają zadane w tym wierszu kryteria.

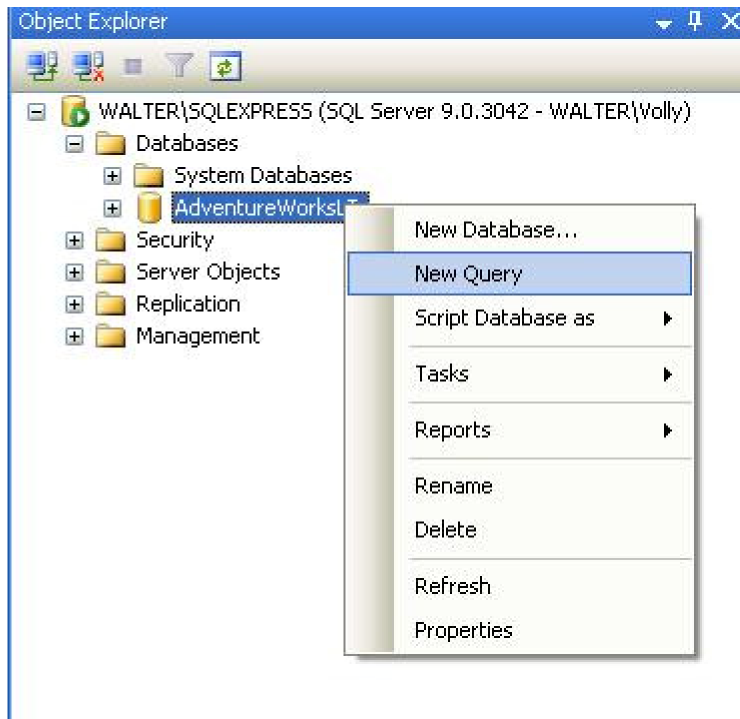
Za chwilę wydamy pierwsze zapytanie do bazy danych.

Ćwiczenie 1. Wydawanie poleceń do *SQL Server*.

1. Uruchom *SQL Server Management Studio* i połącz się z lokalnym serwerem *SQL Server Express*;

2. Otwórz folder *Databases* w oknie *Object Explorer*;

3. Kliknij prawym klawiszem myszy na bazie danych *AdventureWorksLT* i z menu wybierz polecenie *New Query*. W prawej części programu *SSMS* pojawiło się okno z pulsującym kursorem oznaczającym gotowość do przyjęcia zapytań. W podobny sposób można otworzyć wiele okien zawierających zapytania do tej samej lub wielu baz danych;

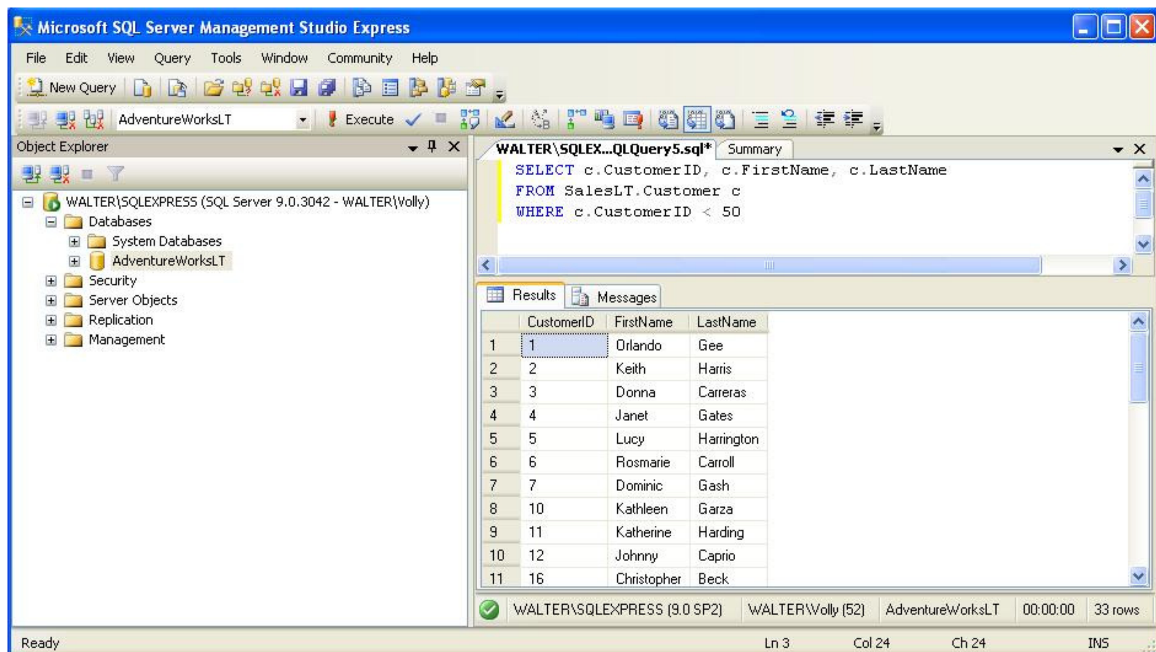


4. Ponieważ nowe okno poleceń otwarliśmy w kontekście bazy *AdventureWorksLT*, to polecenia skierowane są do niej. Jednakże korzystając z listy rozwijanej nad oknem *Object Explorer* możemy zmienić bazę, do której będziemy wysyłać polecenia;

5. W oknie poleceń wpisz następujące zapytanie:

```
SELECT c.CustomerID, c.FirstName, c.LastName
FROM SalesLT.Customer c
WHERE c.CustomerID < 50
```

6. Następnie naciśnij klawisz *F5* (lub przycisk *Execute* z paska narzędzi) w celu wykonania zapytania. Uwaga! Można też użyć kombinacji klawiszy *Ctrl + F5* w celu sprawdzenia poprawności składni polecenia bez jego wykonywania.



7. Polecenie zostało wykonane, a poniżej został wyświetlony wynik, czyli tabela zawierająca żądane informacje. Poniżej tabeli mamy też pasek statusu, dzięki któremu możemy się szybko dowiedzieć jak długo wykonywało się polecenie i ile wzróciło wierszy;

8. Wróćmy do zapytania i wyniku. Wydane zapytanie można byłoby przeczytać w następujący: Wybierz identyfiakor klienta, imię i nazwisko z tabeli przechowyjącej klientów, jednakże interesują nas tylko klienci o identyfikatorach mniejszych od 50;

9. Zwróć też uwagę na charakterystyczny w tym zapytaniu znak „c”. Pisząc go za nazwą tabeli nadajemy jej długiej nazwie krótki alias, dzięki któremu możemy się do niej odwoływać. W tak prostym zapytaniu aliasowanie nie jest konieczne i zapytanie:

   ```
   SELECT CustomerID, FirstName, LastName
   FROM SalesLT.Customer
   WHERE CustomerID < 50
   ```

   da dokładanie taki sam wynik. Jednakże aliasowanie w przyszłości okaże się bardzo dobrą techniką pisania zwięzłych zapytań, więc warto już teraz się do niego przyzwyczajać. Zabieg ten pozwala także uniknąć w przyszłości wielu błędów przy rozbudowie zapytań. Są też przypadki, w których aliasowanie jest konieczne – przykłady takie zopbaczymy przy wyciąganiu danych z wielu tabel.

10. Raz napisane polecenie możemy zapisać do pliku **.sql;*

11. Z menu *File* wybierz polecenie *Save <nazwa okna>.sql* (lub wciśnij *Crtl + S*, albo kliknij ikonę dyskietki w pasku narzędzi;

12. Korzystając ze standardowego okna zapisu plików *Windows* zapisz polecenie na pulpicie jako plik o nazwie *pierwsze.sql*;

13. Używając krzyżyka znajdującego się w prawym górnym rogu okna poleceń zamknij aktualne okno z wprowadzonym poleceniem;

14. Aby wczytać zapisany wcześniej plik wybierz z menu *File* polecenie *Open*, a następnie *File* (lub wciśnij *Ctrl + O*, albo ikonę otwieranego folderu z paska narzędzi); otwórz poprzednio zapisany na pulpicie plik; Jego zawartość pojawiła się w oknie zapytań;

15. Pod istniejącym już poleceniem *SELECT* dopisz nowe, wybierające wszystkie dane z tabeli *Product*:

```
SELECT *
FROM SalesLT.Product
```

16. Wykonaj polecenia (wciskają *F5)*. Zauważmy, że dwa polecenia zostały wykonane równocześnie i okno wyników zawiera dwie tabele;

17. Przy pomocy myszki zaznacz tylko drugie polecenie i naciśnij *F5*; Zauważ, że wykonany został jedynie fragment, który wcześniej został zaznaczony;

18. Wyczyść onko usuwając treść obu zapytań.

Przyjrzyjmy się teraz przykładom zapytań, które opisują możliwości prostego zapytania *SELECT ... FROM ... WHERE*.

Przykład 1. Zmiana nazwy kolumny w wyniku.

```
SELECT c.CustomerID AS KlientID, c.FirstName AS Imie, c.LastName AS Nazwisko
FROM SalesLT.Customer c
WHERE c.CustomerID < 50
```

Słowo kluczowe *AS* nakazuje zamienić domyślną nazwę kolumny wyniku, która jest nazwą kolumny w tabeli, na nową podaną za nim. Taki mechanizm można zastosować w przypadku gdy baza jest w jednym języku, a my robimy raporty dla klientów z wielu państw.

Przykład 2. Tworzenie kolumn wyliczanych

```
SELECT ProductID, OrderQty, UnitPrice, OrderQty * UnitPrice AS Value
FROM SalesLT.SalesOrderDetail
```

W sposób powyższy tworzymy nową kolumnę wyniku, która nie istnieje w fizycznej tabeli. W tym przykładzie w każdym wierszu uzyskamy dodatkową wartość będącą iloczynem wartości z kolumn *OrderQty* i *UnitPrice* z tego wiersza. Zamiast znaku mnożenia możemy użyć dowolnego operatora arytmetycznego oraz możemy budować wyrażenia składające się z więcej niż 2 kolumn. Możemy także używać stałych. Aby obliczyć wartość podatku VAT możemy napisać:

```
SELECT ProductID, OrderQty * UnitPrice AS Value, OrderQty * UnitPrice * 0.22 AS Tax
FROM SalesLT.SalesOrderDetail
```

Oto zestawienie oeratorów arytmetycznych możliwych do użycia:

Operator	Opis
+ (*Dodaj*)	Dodawanie
- (*Odejmij*)	Odejmowanie
* (Pomnóż)	Mnożenie
/ (*Podziel*)	Dzielenie. Uwaga! Dzielenie automatycznie zaokrągla wynik, jeśli dzielimy liczby całkowite. A więc wynik wyrażenia 5/2 to 2. Więcej na ten temat dowiemy sięz rozdziału o typach danych i konwersjach. W przypadku liczb zmiennoprzecinkowych nie ma takiego problemu, więc 3.4 / 2.8 = 1.214285.
% (*Modulo*)	Jest to reszta z dzielenia całkowitego. Na przykład 5 % 2 = 1, bo 2 mieści się w 5 dwa razy i zostaje jeszcze 1 reszty.

W przypadku kolumn wyliczanych konieczne jest nadanie im nazw za pomocą słowa kluczowego *AS*. Jeśli tego nie zrobimy, to kolumna nie będzie miała nazwy.

Istnieje jeszcze jeden operator bardzo nam przydatny, lecz nie działa on na liczbach, lecz na napisach i służy do konkatenacji (łączenia) napisów:

```sql
SELECT c.CustomerID, c.FirstName, c.LastName, c.LastName + ', ' + c.FirstName AS Name
FROM SalesLT.Customer c
```

Jest to ten sam znak plusa, który dodawał dwie liczby. Mechanizm *SQL Servera* sam rozpoznaje czy kolumn są typu liczbowego czy znakowego (napisy) i wybiera odpowiednie działanie.

Przykład 3. Generowanie kolumn ze stałymi

Czasami potrzebujemy wygenerować kolumny ze stałymi. Robimy to w następujący sposób:

```sql
SELECT c.CustomerID, NULL AS KolumnaZNullami, 5.0 AS KolumnaZPiatkami, 'napis' AS
KolumnaZNapisami
FROM SalesLT.Customer c
WHERE c.CustomerID < 50
```

Przykład 4. Logiczne łączenie warunków

W powyższych zapytaniach w klauzuli *WHERE* znajdował się tylko jeden warunek. W większości sytuacji jest on niewystarczający, dlatego język T-SQL pozwala nam łączyć wiele warunków za pomocą operatorów logicznych umieszczonych w poniższej tabeli:

Operator	Znaczenie
AND	Koniunkcja, logicze „i”. Wyrażenie p *AND* q jest prawdziwe wtedy i tylko wtedy, gdy zarówno p i q są wyrażeniami prawdziwymi.
OR	Alternatywa, logiczne „lub”. Wyrażenie p *OR* q jest prawdziwy wtedy i tylko wtedy, gdy chociaż jedno z wyrażeń p, q jest prawdziwe.
NOT	Negacja. Wyrażenie *NOT* p jest prawdziwe wtedy i tylko wtedy, gdy p jest fałszywe.

Następne zestawienie obejmuje operatory porównania dwóch wartości liczbowych:

Operator	Zanaczenie
=	Równe
<	Mniejsze
<=	Mniejsze lub równe
>	Większe
>=	Większe lub równe
<>	Różne

Wyciągnijmy więc z bazy produkty, które mają nieparzysty identyfikator albo są koloru czerwonego i jednocześnie rozmiaru 58.

```
SELECT p.ProductID, p.Color, p.Size
FROM SalesLT.Product p
WHERE p.ProductID % 2 = 0 OR (p.Color = 'Red' AND p.Size = '58')
```

Nawiasy nie są konieczne, lecz warto je stosować, bo w przypadku bardziej skomplikowanych warunków łatwo możemy się pogubić.

Przykład 5. Porównywanie napisów

Proste porównywanie napisów poznaliśmy już w poprzednim przykładzie. Można napisać w warunku, że napisy mają być równe, różne lub nawet mniejsze albo większe. W przypadku tych dwóch ostatnich operatorów decyduje kolejność leksykograficzna, czyli taka, jaką mamy w słowniku. Poza standardowymi operatorami w przypadku napisów mamy dodatkowy operator *LIKE*. Znajdźmy więc produkty, które w swojej nazwie zawierają podsłowo „*ain*” lub takie które mają pięcioliterową nazwę koloru i druga litera to nie „*l*”:

```
SELECT p.ProductID, p.Name, p.Color
FROM SalesLT.Product p
WHERE p.Name LIKE '%ain%' OR p.Color NOT LIKE '_l___'
```

Więcej ciekawych przykładów poznamy przy omawianiu funkcji operujących na napisach.

Przykład 6. Proste porównywanie dat

Daty, podobnie jak napisy można porównywać używając standardowych operatorów porównania. Spróbujmy więc wybrać informacje o powiązaniach klientów z adresami, które uległy zmianie pomiędzy *23 kwietnia 2003 roku*, a *30 lipca 2003 roku*:

```
SELECT *
FROM SalesLT.CustomerAddress ca
WHERE ca.ModifiedDate >= '2003-04-23' AND ca.ModifiedDate =< '2003-07-30'
```

Przykład 7. Porównywanie zakresów

Często zachodzi konieczność sprawdzenia, czy wartość w pewnej kolumny znajduje się w danym zakresie. Do tej pory robiliśmy to w sposób taki jak w przykładzie 6, czyli za pomocą dwóch operatorów: >= i <=. Przyjrzyjmy się teraz użyciu operatora *BETWEEN* dającego taki sam rezultat.

```
SELECT *
FROM SalesLT.CustomerAddress ca
WHERE ca.ModifiedDate BETWEEN '2003-04-23' AND '2003-07-30'
```

Operatorem *BETWEEN* można porównywać dowolne wielkości: daty, liczby, itp. Należy pamiętać, że podane w zapytaniu ograniczenia zakresu, to końce przedziału zamkniętego.

Przykład 8. Usuwanie dubli z wyniku

Wykonajmy następujące zapytanie, chcąc zobaczyć wszystkie możliwe kolory produktów:

```
SELECT p.Color
FROM SalesLT.Product p
```

Zostało zwróconych mnóstwo wierszy, lecz kolory bardzo często się powtarzają. Użyjemy teraz polecenia *DISTINCT* aby wyświetlić tylko unikalne wartości:

```
SELECT DISTINCT p.Color
FROM SalesLT.Product p
```

Teraz każdy z kolorów jest wyświetlony tylko raz.

Przykład 9. Pobieranie części wyniku

Zdarza się również dosyć często, że chcemy pobrać jedynie część wyniku. Umożliwia nam to polecenie *TOP*. Można go użyć na dwa sposoby: wskazać konkretną liczbę rekordów jaką chcemy pobrać, bądź procentową ilość wierszy wyniku. Pobierzmy więc 30 pierwszych produktów z tabeli *Product*:

```
SELECT TOP 30 *
FROM SalesLT.Product p
```

Następnie pobierzmy 40% tej tabeli:

```
SELECT TOP 40 PERCENT *
FROM SalesLT.Product p
```

Zadania

1. Używając polecenia *SELECT* wybierz dane ze wszystkich kolumn i wszystkich wierszy tabeli *Product*.
2. Korzystając z dowolnej tabeli w bazie *AdventureWorksLT* napisz zapytanie które zwróci kolumnę 15 wartości *NULL*.
3. Czy kolumny występujące w warunku *WHERE* muszą być uwzględnione w linii *SELECT*?
4. Dlaczego w przykładzie 4. kolumna rozmiar jest przyrównana do napisu '58', a nie do liczby 58?
5. Znajdź wszystkie produkty, które są koloru czerwonego lub niebieskiego.
6. Wybierz wszystkie dane klienta o identyfikatorze 42.
7. Wybierz imiona i nazwiska wszystkich klientów.
8. Wybierz wszystkie produkty, które są rozmiaru L lub większego, tzn. XL, XXL, itd.
9. Wybierz te powiązania klientów z adresami, które zostały zmodyfikowane po dniu 1 sierpnia 2002 r.
10. Wybierz wszystkie adresy o identyfikatorach większych bądź równych 604 i mniejszych od 616 używając operatora *BETWEEN*. Zmodyfikuj zapytanie tak, aby zwracając wynik jak poprzednio nie używało operatora *BETWEEN*.
11. Wybierz wszystkie produkty, których kolor nie jest czerwony.
12. Wybierz wszystkie produkty, których kolor nie jest czerwony, a cena (*ListPrice*) jest większ równa niż 9,50 i mniejsza od 390.
13. Co się stanie gdy będziemy dodawać liczbę do napisu?
14. Wybierz wszystkie nazwy miast występujące w adresach z bazy tak, aby każda z nich pojawiła sie tylko jeden raz.
15. Wybierz wszystkie pary kraju i miasta występujące w adresach z bazy tak, aby każda z nich pojawiła sie tylko jeden raz.
16. Wybierz 3 dowolne produkty koloru niebieskiego.

17. Wybierz 80% produktów koloru czerwonego.
18. Utwórz wynik zapytania, który zwróci tylko jedną kolumnę o nazwie „cztery” i 10 wierszy. Każda komórka powinna zawierać liczbę 4. Dodatkowo znak ‘4’ nie może występować w treści zapytania. *Wskazówka: Użyj kolumny wyliczanej*.
19. Budując trzy kolejne zapytania sprawdź w jakim mieście mieszka klient Brian Groth.
    Wskazówka: Zaczynając od tablicy z klientami przejdź do tabeli z adresami korzystając z relacji pomiędzy tymi tabelami.

Sortowanie wyniku zapytania

Dotychczas wyniki były zwracane w kolejności od nas niezależnej. Dla przykładu wybierzmy sobie kilka kolumn z tabeli produkt:

```
SELECT p.ProductID, p.Name, p.ListPrice, p.Color
FROM SalesLT.Product p
```

Zauważmy, że wynik jest posortowany po klumnie *ProductID*, choć nie zawsze musi tak być i dopóki nie użyjemy jawnego nakazu sortowania, to kolejność wierszy jest przypadkowa. Nasz wynik jest posortowany w kolejności wstawiania danych do tabeli. Co zrobić więc gdy chcemy posortować wiersze według własnego życzenia? W takim wypadku należy użyć dodatkowej klauzuli o nazwie *ORDER BY*. Posortujmy więc poprzedni wynik tak, aby kolumna *Name* była posortowana rosnąco i ograniczmy wyświetlane produkty tylko do takich, których identyfikator należy do przedziału (712,800]:

```
SELECT p.ProductID, p.Name, p.ListPrice, p.Color
FROM SalesLT.Product p
WHERE p.ProductID BETWEEN 711 AND 800
ORDER BY p.Name
```

Klauzulę *ORDER BY* umieszczamy poniżej klauzuli *WHERE* i bezpośrednio za nią wypisujemy kolumny po których chcemy sortować. W poprzednim zdaniu napisaliśmy „kolumny", bo faktycznie można sortować po wielu kolumnach – należy wówczas wypisać je w klauzuli *ORDER BY* w kolejności ważności. Co więcej możemy również sortować malejąco, dopisując za nazwą kolumny, którą chcemy sortować w taki sposób słowo *DESC*:

```
SELECT p.ProductID, p.Name, p.ListPrice, p.Color
FROM SalesLT.Product p
WHERE p.ProductID BETWEEN 711 AND 800
ORDER BY p.ListPrice, p.ProductID DESC
```

Posortowaliśmy produkty od najtańszego do najdroższego, a gdy cena jest jednakowa bierzemy pod uwagę drugie kryterium, jakim jest identyfikator produktu. Co więcej, wewnątrz tej samej ceny produkty posortowane są malejącymi identyfikatorami.

Wiemy, że w klauzuli *ORDER BY* należy użyć nazwy kolumny po której chcemy sortować. Spróbujmy więc wykonać polecenie:

```
SELECT p.ProductID, p.ListPrice * 0.22 AS TaxedPrice
FROM SalesLT.Product p
WHERE p.ProductID BETWEEN 711 AND 800
ORDER BY TaxedPrice
```

Możemy rózwnież to zapytanie napisać nastepująco:

```
SELECT p.ProductID, p.ListPrice * 0.22 AS TaxedPrice
FROM SalesLT.Product p
WHERE p.ProductID BETWEEN 711 AND 800
ORDER BY p.ListPrice * 0.22
```

Albo tak, wskazując, że sortujemy po drugiej kolumnie wyniku:

```
SELECT p.ProductID, p.ListPrice * 0.22 AS TaxedPrice
FROM SalesLT.Product p
WHERE p.ProductID BETWEEN 711 AND 800
ORDER BY 2
```

Czasami jednak chcemy osiągnąć efekt odwrotny do sortowania, gdy na przykład chcemy pobrać dane o 4 losowo wybranych produktach. Robimy to w następujący sposób:

```
SELECT TOP 4 p.*
FROM SalesLT.Product p
ORDER BY NEWID()
```

Zauważmy, że za każdym wykonaniem wynik jest inny.

Wróćmy teraz do porządkujących właściwości polecenia *ORDER BY*. Łącząc to polecenie z funkcją *ROW_NUMBER()* możemy ponumerować kolejne wiersze:

```
SELECT soh.SalesOrderID, soh.CustomerID,
       ROW_NUMBER() OVER (ORDER BY soh.SalesOrderID) AS Number
FROM SalesLT.SalesOrderHeader soh
WHERE soh.SalesOrderID > 10000
```

Zadania

1. Wybierz z bazy wszystkie adresy sortując je rosnąco względem nazwy miejscowości.
2. Wybierz z bazy imiona i nazwiska klientów sortując je rosnąco względem imienia, a jeśli imiona są takie same, to malejąco względem nazwiska.
3. Znajdź najdroższy produkt w bazie.
4. Znajdź naj lżejszy produkt w bazie.
5. Wybierz 30 procent produktów, które są najdroższe.
6. Wybierz 10 losowych produktów koloru czerwonego lub biebieskiego i rozmiaru L lub XL.
7. Dla każdej ceny produktu (*ListPrice*) oblicz wartość podatku VAT 22% i zwróć wszystkie nazwy produktów wraz z obliczoną wartością sortując malejąco względem obliczonego podatku.
8. Zwróć nazwy wszystkich niebieskich produktów numerując je kolejno w dodatkowej kolumne o nazwie *Lp*. zaczynając od 1.
9. Używając operatora modulo i dowolnej tabeli z naszej bazy wygeneruj 8 następujących wierszy:

   ```
   0 1
   1 0
   0 1
   1 0
   0 1
   1 0
   0 1
   1 0
   ```

   Dokonaj jak najmniejszej modyfikacji w klauzuli *ORDER BY*, tak aby odwórcić wynik symetrycznie w pionie.

Praca z wartościami typu NULL, funkcje ISNULL oraz NULLIF

Rzeczą, która może sprawiać trochę kłopotów są wartości puste (*NULL*). Przykład tych wartości można zobaczyć wokonując poniższe polecenie i obejrzenie kolumny *AddressLine2*:

```
SELECT *
FROM SalesLT.Address
```

Problemy mogą być następujące. Jeśli przykładowo wybieramy z tabeli te adresy, które mają miasto różne od Wrocławia i piszemy:

```
WHERE miasto <> 'Wrocław'
```

to w wyniku nie będzie tych adresów, które nie mają uzupełnionego pola miasto. Ale przecież wartość pusta jest różna od „*Wrocław*" więc, adres bez miasta też powinien być zwrócony w wyniku. Niestety standardowe operatory porównania na wartościach pustych działają według skomplikowanego, choć przewidywalnego schematu.

Na szczęście z pomocą przychodzi nam dodatkowe porównanie *IS NULL* stosowane często z negacją (*IS NOT NULL*). Porównanie to sprawdza czy dana wartość jest pusta czy nie. Wróćmy do naszego pierwotnego warunku. Jeśli wiemy, że w kolumnie miasto może znajdować się wartość pusta i chcemy, żeby faktycznie była ona różna od wartości *'Wrocław'*, to powinniśmy napisać:

```
WHERE miasto <> 'Wrocław' OR miasto IS NULL
```

Tak więc powinniśmy być ostrożni w przypadkach, gdy w porównaniu arytmetycznym po jednej ze stron może występować wartość *NULL*.

Omówimy teraz dwie ciekawe funkcje związane z wartościami pustymi. Pierwszą z nich jest ***ISNULL***. Bierze ona dwa parametry i działa w sposób taki, że jeśli pierwszy z nich jest wartością pustą, to zwraca wartość podaną jako drugi parametr. Przykład: wyświetlmy wszystkie adresy, a dokładniej pierwszą i drugą linię adresu, kod pocztowy i miasto. W naszej przykładowej bazie druga linia adresu jest często wartością pustą. Zamiast wyświetlać *NULL*-e wolelibyśmy tekst 'brak drugiej lini adresu'. Oto jak to zrobić:

```
SELECT a.AddressLine1, ISNULL(a.AddressLine2, 'brak drugiej lini adresu') AS AddressLine2,
a.PostalCode, a.City
FROM SalesLT.Address a
```

Funkcja powyższa przydaje się bardzo często, więc warto ją zapamiętać. Inną funkcją, rzadziej używaną, jest ***NULLIF***. Bierze ona dwa paramerty i zwraca pierwszy jeśli podane parametry są różne, albo zwraca *NULL* jeśli podane parametry są równe. Możemy się posłużyć tą funkcją, aby sprawdzić czy w tabeli dobrze są pododawane kolumny:

```
SELECT soh.SubTotal, soh.TaxAmt, soh.Freight, soh.TotalDue, NULLIF(soh.TotalDue, soh.SubTotal
+ soh.TaxAmt + soh.Freight) AS OK
FROM SalesLT.SalesOrderHeader soh
```

Pozyższe zapytanie wyświetli w dodatkowej kolumnie wartości *NULL*, jeśli suma w kolumnie *TotalDue* jest dobrze policzona i wartość *TotalDue*, jeśli jest błąd.

Zadania

1. Wybierz te adresy z bazy, które nie mają wartości w kolumnie *AddressLine2*.
2. Wybierz te produkty, które są aktualnie w sprzedaży (kolumna *SellEndDate* jest pusta).
3. Wybierz te produkty, których sprzedaż została zakończona.
4. Wybierz te produkty, które nie mają informacji o wadze, ale mają informacje o rozmiarze i są koloru czarnego.
5. Jeśli w poprzednim zadaniu użyto operatora *IS NOT NULL*, to zmień treść zapytania tak, aby tego operatora nie zawierała, a w przeciwnym wypadku zmodyfikuj zapytanie tak, aby operator *IS NOT NULL* został użyty.
6. Wybierz listę wszystkich produktów w taki sposób, że jeśli produkt nie zawiera informacji o wadze lub rozmiarze, to w wyniku powinn pojawić się zero.
7. Wybierz informacje o tych produktach, których podwojona waga jest większa od ceny (*ListPrice*). Jeśli produkt nie ma podanej wagi, należy przyjąć wartość 500 jako tę wagę.
8. Wbierz wszystkie informacje o adresach z miasta *Toronto*. Jeśli nie ma informacji o drugiej lini adresu, to w komórce powinien pojawić się napis '*brak informacji*'.
9. Wybierz następujące informacje o adresach ze stanów zjednoczonych: identyfikator, miasto oraz dwie linie adresu rozdzielone przecinkiem i spacją jako jedna kolumna. Jeśli druga linia adresu nie zawiera żadnych informacji, to w wynikowej kolmnie z adresem powinna znaleźć się tylko jedna linia (bez przecinka i spacji). Wskazówka: jeśli dodamy napis do wartości pustej, to w wyniku otrzymamy wartość pustą.
10. Wybierz z bazy kategorie główne produktów, tzn. te, które nie są w żadnej innej kategorii.
11. Wyświetl tabelę zawieracjącą wszystkie adresy. Wszystkie wystąpienia słowa *Canada* w kolumnie *CountryRegion* powinny być zastąpione wartością pustą.
12. Wyświetl następujące informacjo o produktach: nazwa i cena. Jednakże jeśli cena produktu jest równa 49.99, to nie wyświetlaj informacji o cenie, ale nazwę tego produktu należy wyświetlić.

Złączenia wewnętrzne i zewnętrzne

Do tej pory mogliśmy wyciągać tylko dane z pojedynczej tabeli. Teraz dzięki złączeniom tabel według logiki relacji będziemy tworzyć rozbudowane zapytania wyciągające dane z wielu tabel. Wyróżniamy 4 podstawowe typy złączeń: *CROSS JOIN*, *INNER JOIN*, *LEFT (RIGHT) OUTER JOIN* oraz *FULL OUTER JOIN*.

Złączenie typu ***CROSS JOIN*** jest iloczynem kartezjańskim dwóch zbiorów reprezentowanych przez złączane tabele. Spójrzmy na przykład połączenia tabel *R* i *S* (*r1* i *s1* skrótowo reprezentują wszystkie kolumny tabel *R* i *S*):

R

r1
2
3

S

s1
2
1
4

R INNER JOIN S

r1	s1
2	2
2	1
2	4
3	2
3	1
3	4

Aby utworzyć tak połączoną tabelę musimy wydać następujące polecenie:

```
SELECT R.r1, S.s1
FROM R CROSS JOIN S
```

Oczywiście za klauzulą *FROM* możemy dopisać poznane już klauzule *WHERE* i *ORDER BY*.

Z definicji iloczynu kartezjańskiego wynika, że w ilość wierszy w tabeli wynikowej jest równa iloczynowi ilości wierszy w tabelach łączonych. I tu pojawia się problem, bo przyłączeniu większych tabel rozmiar wyniku jest ogromny. *CROSS JOIN* to jeden z łatwiejszych sposobów na zatrzymanie pracy serwera SQL. Złączenie tego typu więc raczej się nie używa, choć są przypadki, kiedy okazuje się to złączenie przydatne. Przykłady użycia:

- Połączenie dwóch jednowierszowych table ze sobą, aby utworzyć szerszą tabelę jednowierszową;
- Szybkie generowanie kolumn zawierających liczby naturalne z zadanego przedziału.

Generalnie wszystkie następne złączenia jesteśmy w stanie zrealizować złączeniem *CROSS JOIN* oraz manipulacją warunkiem *WHERE*, jednakże z uwagi na silne obciążenie serwera w przypadku pomyłki

w pisaniu zapytania, odradza się stanowczo korzystania z tego złączenia, jeśli nie zachodzi konieczność.

Następne złączenie to złączenie wewnętrzne – ***INNER JOIN***. Zwraca ono tylko te wiersze z dwóch tabel, które pasują do siebie względem kolumny (kolumn) łączenia. Najlepiej zilustruje to przykład:

```
SELECT p.ProductID, p.Name, p.ProductCategoryID, pc.ProductCategoryID, pc.Name
FROM
       SalesLT.Product p
              INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
```

Aby połączyć tabele *Product* i *ProductCategory* użyliśmy istniejącej pomiędzy tymi tabelami relacji „jest w". Złączenie nastąpiło po kolumnie *ProductCategoryID*, która jest kluczem głównym w tabeli *ProductCategoryID* oraz kluczem obcym w tabeli *ProductID*. Kolumny *ProductCategoryID* wyświetliliśmy, aby łatwo było widać jak wiersze się dopasowały. Z reguły identyfikatorów nie wyświetla sie w raportach. A więc gdybyśmy zostali poproszeni o wybranie z bazy produktów o cenie mniejszej niż 234,00 razem z ich katregoriami, a całość posortowana względem malejących cen, to moglibyśmy napisać:

```
SELECT pc.Name AS Category, p.Name, p.ListPrice
FROM
       SalesLT.Product p
              INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
WHERE p.ListPrice < 234.0
ORDER BY p.ListPrice DESC
```

Na przykładzie powyższego zapytania po raz pierwszy widać dlaczego trzeba aliasować tabele. Po pierwsze zamiast całej nazwy tabeli możemy używać aliasów po słowie kluczowym *ON*. A po drugie obie tabele mają kolumnę o nazwie Name. Bez użycia aliasu polecenie *SELECT* nie wiedziałoby którą kolumnę wstawić. Zauważmy też, że w tabeli z kategoriami, każda z kategorii występuje dokładnie jeden raz, a w wyniku zapytania występuje ona tyle razy ile razy się dopasowała do produktu.

Zajmijmy się teraz złączeniami zewnętrznymi, a konkretniej złączeniem *LEFT OUTER JOIN* (w skrócie ***LEFT JOIN***). Działa ono następująco: bierze wszystkie wskazane kolumny z tabeli stojącej po lewej stronie słowa kluczowego *LEFT JOIN* i próbuje dopasować do nich po klumnie łączenia wiersze z tabeli stojącej po prawej stronie. Jeśli się to uda, to do wiersza wstawiane są wartości z prawej kolumny. Jeśli natomiast rekord z lewej tabeli nie ma swojego odpowiednia w prawej tabeli, to we wskazane kolumny z prawej tabeli wpisana jest wartość *NULL*. Dla przykładu wybierzmy sobie wszystkie adresy i sprawdźmy kiedy był pod każdy z nich wysłany towar:

```
SELECT
       a.AddressLine1, a.AddressLine2, a.City, a.PostalCode, soh.ShipDate
FROM
       SalesLT.Address a
              LEFT JOIN SalesLT.SalesOrderHeader soh ON a.AddressID = soh.ShipToAddressID
```

W wyniku zapytania otrzymaliśmy pełną listę adresów. W kolumnie *ShipDate* jest data wysyłki, jeśli ta miała miejsce, a jeśli pod dany adres nic nie wysyłono to jest wartość *NULL*. Druga sytuacja ma miejsce, gdy identyfikator adresu nie występuje w tabeli z zamówieniam – nie może się wtedy dopasować.

Złączenie typu ***RIGHT JOIN*** jest symetryczne do *LEFT JOIN* i wskazuje, że to prawa tabela ma być wyświetlona w całości, a lewa ma być dopasowywana. Z reguły używa się raczej polecenia *LEFT JOIN*. Taki sam efekt jak powyżej można uzyskać zamieniając miejscami tabele *Address* i *SalesOrderHeader* oraz zastępując frazę *LEFT JOIN* przez *RIGHT JOIN*.

Ostatnim, a zarazem dość ciekawym typem złączenia zewnętrznego jest ***FULL OUTER JOIN***. Zwraca ono wiersze które się dopasowały, wiersze które istnieją w lewej tabeli a nie istnieją w prawej oraz te, które istnieją w prawej, a nie istnieją w lewej. Można więc powiedzieć, że są to trzy typy złączeń (*INNER*, *LEFT* i *RIGHT*) zamknięte w jednym poleceniu. Złączenia tego używa się bardzo rzadko.

Oczywiście złączenia nie dotyczą tylko dwóch tabel. Przykładem konieczności użycia więcej niż dwóch tabel jest złączenie tabel reprezentujących relację wiele do wielu. Jak pamiętamy realizowana jest ona poprzez dodatkową tabelę. Połączmy więc klientów z ich adresami:

```
SELECT
        a.AddressLine1, a.City, a.PostalCode, c.FirstName, c.LastName
FROM
        SalesLT.Customer c
                INNER JOIN SalesLT.CustomerAddress ca ON c.CustomerID = ca.CustomerID
                INNER JOIN SalesLT.Address a ON ca.AddressID = a.AddressID
```

Należy jeszcze wspomnieć na koniec, że można dowolnie łączyć wymienione typy złączeń w jednym zapytaniu, a warunki złączenia po słowie kluczowym *ON* mogą być bardziej skomplikowane, podobne do tych z klauzuli *WHERE*. Oto przykład:

```
SELECT
        p.Name, pc.Name AS Category
FROM
        SalesLT.Product p
                LEFT JOIN SalesLT.SalesOrderDetail sod ON p.ProductID = sod.ProductID AND
sod.SalesOrderDetailID IS NULL
                INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
```

Zaytanie wybiera produkty, które nigdy nie były zamawiane, wraz z ich kategoriami.

Za nami złączenia, czyli najbardziej istotna część kursu.

Zadania

1. Wyświetl nazwy wszystkich czerwonych produktów wraz z nazwą ich kategroii i nazwą modelu.
2. Wybierz następujące informacje dla wszystkich porduktów: nazwa kategorii, nazwa podkategorii oraz nazwa i cena produktu.
3. Dla każdego klienta wyświetl jego wszystkie adresy nie używając operatora CROSS JOIN.
4. Przepisz powyższe zapytanie tak, aby używało operatora CROSS JOIN, ale dawało taki sam wynik w równie szybkim czasie.
5. Za pomocą operatora CROSS JOIN połącz tabele Address i Customer wybierając z nich wszystkie informacje. Ile wierszy zwróciło zapytanie i jak długo trwało?
6. Do powyższego złączenia dodaj jeszcze tabelę Product łącz ją z dwiema pozostałymi równiez operatorem CROSS JOIN? Ile wierzy zwróciło zapytanie i jak długo trwało? Jeśli nie starczy Ci cierpliwości lub moc komputera nie pozwoli na wykonanie tego zapytania do końca, to policz ile teoretycznie wierszy będzie zawieral wynik.

7. Używając operatora CROSS JOIN utwórz tabelę zawierającą w pierwszej kolumnie imię i nazwisko klienta o identyfikatorze 18, a w drugiej nazwę miasta z adresu o ID = 9.
8. Używając złączenia LEFT JOIN wybierz wszystkie adresy, oraz informacje o zamówieniach, których rachunki zostały wysłane na te adresy. Jeśli na adres nie wysłano zadnego zamówienia, to kolumny opisujące zamówienie powinny zawierać wartości puste.
9. Zmodyfikuj zapytanie z poprzednioego zadania tak, aby zastąpić operator LEFT JOIN przez RIGHT JOIN, a wynik pozostał taki sam.
10. Nie zmieniając typu złączenia zmodyfikuj zapytanie będące wynikiem zadania poprzedniego tak, aby wynik nie zawierał adresów, na które nie został nigdy wysłany rachunek.
11. Napisz przynajmniej jeden przykład dotyczący bazy AdventureWorksLT, kiedy wykorzystałbyś złączenie FULL OUTER JOIN.
12. Wyświetl wszystkie czerwone produkty wraz z nazwą modelu i jego opisem w języku francuskim. Wynik powinien być posortowany rosnąco ze względu na cenę, a jeśli cena jest taka sama to rosnąco ze względu na opis modelu.
13. Znajdź wszystkich klientów, którzy zamawiali produkty z kategorii 'Bikes'.
14. Znajdź wszystkie adresy, na które zostały dostarczone produkty z kategorii 'Clothing'.
15. Wybierz te produkty, które nigdy nie zostały dostarczone do Kanady.
16. Przy użyciu jednego zapytania znajdź te adresy, które nie są przyporządkowane (przez tabelę CustomerAddress do żadnego klienta i tych klientów, którzy nie posiadają żadnego adresu.
17. Wybierz te produkty, które nie zostały nigdy zamówione.
18. Wybierz nazwy produktów, które zostały kiedykolwiek zamówione przez Waltera Maysa. Tabela powinna zawierać również informacje, na jaki adres zostało wysłane zamówienie i na jaki adres zostal wysłany rachunek.
19. Wybierz wszystkich klientów z USA i dla każdego z nich wybierz nazwy wszystkich kategorii z których produkty zostały im dostarczone.
20. Wybierz klientów nie mieszkających w USA, który zamówili dowolny czerwony produkt lub produkt z kategorii 'Bike'.

Podzapytania nieskorelowane i skorelowane

Kolejny mechanizm, dzięki któremu wzrastają znacząco możliwości języka T-SQL to podzapytania. Dają one możliwość użycia zapytania *SELECT* wewnątrz innego zapytania *SELECT*, stąd jedno z nich będziemy nazywać zapytaniem zewnętrznym, a drugie zapytaniem wewnętrznym. Oczywiście poziomów zagnieżdżeń może być więcej i dane zapytanie *SELECT* może być zarówno zewnętrzne, jak i wewnętrzne. Spójrzmy na pierwszy przykład, który zwraca tych klientów, którzy posiadają adresy:

```
SELECT *
FROM SalesLT.Customer c
WHERE c.CustomerID IN
(
        SELECT ca.CustomerID
        FROM SalesLT.CustomerAddress ca
)
```

Jak widzimy w zapytaniu są dwa polecenia *SELECT*, a wszystkie nowe dla nas rzeczy dzieją się w klauzuli *WHERE*. Użyliśmy operatora *IN*, który stoi pomiędzy dwoma argumentami: lewy musi być pojedynczą wartością (w tym wypadku identyfikatorem klienta), a drugi zbiorem wartości (w tym wypadku zbiorem identyfikatorów klientów, którzy są przypisani do jakiegoś adresu). Operator ten zwraca prawdę, jeśli element z lewej strony jest w zbiorze z jego prawej strony. Był to najbardziej prosty przykład podzapytania – podzapytanie nieskorelowane. To samo zapytanie można napisać używając podzapytania skorelowanego. Skorelowanie oznacza, że podzapytanie używa jednej lub więcej kolumn z zapytanie zewnętrznego. Poniższe zapytanie jest logicznym odpowiednikiem poprzedniejwersji nieskorelowanej:

```
SELECT *
FROM SalesLT.Customer c
WHERE EXISTS
(
        SELECT *
        FROM SalesLT.CustomerAddress ca
        WHERE c.CustomerID = ca.CustomerID
)
```

W tym przypadku podzapytanie koreluje wartość kolumny zewnętrznego zapytania *CustomerID* z wartością podzapytania *CustomerID*. Orzeczenie *EXISTS* zwraca prawdę, jeśli choć jeden wiersz jest zwracany przez podzapytanie. Podzapytania można również użyć w liście instrukcji *SELECT*. Poniższe zapytanie zwraca wszystkich klientów, którzy mają choć jeden adres i dodatkowo zwraca identyfikator jednego z adresów:

```
SELECT
        c.CustomerID,
        (
                SELECT TOP 1 ca.AddressID
                FROM SalesLT.CustomerAddress ca
                WHERE ca.CustomerID = c.CustomerID
        ) AS AddressID
FROM SalesLT.Customer c
```

Zwróćmy uwagę na użycie polecenia *TOP*. Sprawdźmy, że gdy usuniemy *TOP* z podzapytania, to wykonanie całości zakończy się błędem. Dzieje sie tak dlatego, że pewien z pracowników ma więcej niż jeden adres, czyli podzapytanie zwraca więcej niż jedną linię, a polecenie *SELECT* oczekuje od podzapytania dokładnie jednej wartości.

Jeszcze innym miejscem, gdzie może wystąpić podzapytanie jest klauzula *FROM*. Zamiast podawać nazwę tabeli, możemy ująć w nawiasach nawet bardzo skomplikowane podzapytanie zwracające tabelę, następnie nadać tej tabeli alias i odwoływać się do niej przez ten alias jak robiliśmy uprzednio. Przykład użycia podzapyania w klauzuli *FROM*:

```
SELECT
        a.AddressID, c1.CustomerID, c1.FirstName
FROM
        SalesLT.Address a
                INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
                INNER JOIN
                                (
                                        SELECT c.*
                                        FROM SalesLT.Customer c
                                        WHERE c.FirstName LIKE '%k%') c1 ON ca.CustomerID =
c1.CustomerID
```

Powyższe zapytanie wybiera pary identyfikatorów (klient, adres) ale tylko dla klientów, którzy w pierwszym imieniu mają literę *k*.

Podamy jeszcze pkilka przykładów użycia operatorów, które wykożystują podzapytaniami można ich użyć z w warunku *WHERE*.

Przykład 1. Operator *ANY* (*SOME*).

```
SELECT *
FROM SalesLT.Product p
WHERE p.ListPrice < ANY (SELECT p1.ListPrice FROM SalesLT.Product p1)
```

Powyższe zapytanie zwraca wszystkie produkty z wyjątkiem najdroższego. Innymi słowy zwraca te produkty, które mają cenę mniejszą niż dowolna cenę ze zbioru wszystkich produktów.

Przykład 2. Operator *ALL*.

Odwrotne działanie do *ANY* ma operator *ALL*:

```
SELECT *
FROM SalesLT.Product p
WHERE p.ListPrice <= ALL (SELECT p1.ListPrice FROM SalesLT.Product p1)
```

Zapytanie to zwraca najtańszy produkt (który był poprzednio wykluczony). W warunku *WHERE* mówimy że chcemy wybrać produkty, które mają cenę mniejszą bądź równą cenom wszyskich znanych produktów.

Przykład 3. Porównanie z wartością skalarną zwracaną przez podzapytanie.

Możemy na przykład w dość „wyrafinowany" sposób wybrać produkt o identyfikatorze 762:

```
SELECT p.*
FROM SalesLT.Product p
WHERE p.ProductID = (SELECT 762)
```

Oczywiście podzapytanie może być dowolnie skomplikowane, jednakże należy pamiętać, aby zwracało dokładnie jedną wartość!

Zadania

1. Używając podzapytania nieskorelowanego wybierz te ardesy, na które zostały kiedykolwiek wysłane rachunki.
2. Napisz to samo używając podzapytania skorelowanego.
3. Dla każdego klienta ze *Stanów Zjednoczonych* znajdź nazwę jednego z zamówionych przez niego produktów. Produkt powinien być wybrany losowo i powinien być inny za każdym razem, kiedy wykonujemy zapytanie.
4. Dla każdego z krajów znajdź liczbę jego mieszkańców i liczbę adresów.
5. Znajdź medianę z wartości znajdujących się w kolumnie *ProductID* w tabeli *Product*.
6. Wybierz wszystkie główne kategorie produktów wraz ze średnimi cenami produktów w tych kategoriach. Wynik nie powinien zawierać kategorii, której średnia jest najwyższa.
7. Policz średnią cenę produktów dla każdego z możliwych kolorów. Raport powinien zawierać tylko te kolory, dla których średnia jest mniejsza od średniej ceny wszystkich produktów.
8. Wybierz te kategorie główne, które zawierają więcej produktów niebieskich niż czerwonych lub więcej produktów żółtych niż niebieskich.

Typy danych, konwersje typów

Rodzaj gromadzonych danych ogranicza typ danych, jaki możemy przechowywać w kolumnie, a w niektórych przypadkach róznież ogranicza zakres przyjmowanych wartości. Typy danych w języku Transact-SQL dzielimy na następujące grupy:

1. **Liczbowe dokładne;**
2. **Liczbowe przybliżone;**
3. **Walutowe;**
4. **Data i czas;**
5. **Znakowe;**
6. **Binarne;**
7. **Specjalnego zastosowania** (omówienie tej grupy wykracza poza kurs).

Ad1. Dokładnych typów danych liczbowych używamy do przechowywania danych zawierających zero lub więcej miejsc dziesiętnych po przecinku. Danych tych typów możemy używać we wszystkich operacjach arytmetycznych bez dodatkowych wymagań. Poniższa tabela przedstawia dokładne typy danych:

Typ danych	Zajmowana pamięć	Zakres wartości	Przeznaczenie
bigint	8 bajtów	-2^63 do 2^63-1	Przechowuje duże liczby całkowite, które mogą być dodatnie lub ujemne
int	4 bajty	-2^31 do 2^31-1	Przechowuje liczby całkowite, które mogą być ujemne lub dodatnie
smallint	2 bajty	-32768 do 32767	Przechowuje liczby całkowite, które mogą być ujemne lub dodatnie
tinyint	1 bajt	0 do 255	Przechowuje mały zakres liczb całkowitych dodatnich
decimal(p,s)	5-17 bajtów zależnie od dokładności	-10^38+1 do 10^38 -1	Przechowuje liczby dziesiętne do

			maksymalnie 38 miejsc
numeric(p,s)	5-17 bajtów zależnie od dokładności	-10^38+1 do 10^38 -1	Przechowuje liczby dziesiętne do maksymalnie 38 miejsc

Liczbowe typy danych *numeric* i *decimal* wymgają podania parametrów do pełnego określenia definicji typu. Parametry te definiują skalę i dokładność typu. Na przykład *decimal(12,4)* definiuje watość dziesiętną, która może mieć do 12 cyfr dziesiętnych z czterema miejscami po przecinku.

Ad 2. Przybliżone typy danych liczbowych przechowują wartości dziesiętne. Jednakże dane zapisywane jako typu float lub real są dokładne jedynie z dokładnością określoną w definicji typu. Każde miejsce dziesiętne na prawo od ustalonej pozycji nie ma gwarancji dokładnego zapisu. Na przykład, jeśli liczbę 1,00015454 zapiszemy jako typ float(8), kolumna ta przechowa jedynie 1,000154. Poniższa tabela zawiera zestawienie typów przybliżonych:

Typ danych	Zajmowana pamięć	Zakres wartości	Przeznaczenie
float(p)	4 lub 8 bajtów	-2,23^308 do 2,23^308	Przechowuje duże liczby zmiennoprzecinkowe, które przekraczają zakres typu decimal
real	4 bajty	-3,4^38 do 3,4^38	Ciągle dozwolony, ale zmieniony na float w celu zgodności ze standardem SQL-92

Ad 3. Walutowe typy danych zostały zaprojektowane do przechowywania wartości pieniężnych z czterema miejscami po przecinku. Poniższa tabela przedstawia zestaienie typów wlutowych

Typ danych	Zajmowana pamięć	Zakres wartości	Przeznaczenie
money	8 bajtów	-922 337 203 685 477,5808 do 922 337 203 685 477,5807	Przechowuje duże wartości walutowe
smallmoney	4 bajty	-214 748,3648 do 214 748,3647	Przechowuje małe wartości walutowe

Ad 4. Typy daty i czasu

Poniższa tabela zawiera typy danych służące do przechowywania daty i czasu:

Typ danych	Zajmowana pamięć	Zakres wartości	Przeznaczenie
datetime	8 bajtów	Od 1 stycznia 1753 do 31 grudnia 9999 z dokładnością do 3,33 milisekundy	Przechowuje bardzo dokładny czas
smalldatetime	4 bajty	Od 1 stycznia 1900 do 6 czerwca 2079 z dokładnością do 1 minuty	Przechowuje mniej dokładny czas

Ad 5. Znakowe typy danych

Aby przechowywać dane znakowe (napisy), wybieramy jeden z typów zaprojektowanych w tym celu. Każdy z nich zajmuje jeden lub dwa bajty pamięci na znak, w zależności od tego czy typ używa kodowania ANSI czy Unicode. Poniższa tabla zawiera możliwe typy znakowe:

Typ danych	Zajmowana pamięć	Liczba znaków	Przeznaczenie
char(n)	1 – 8000 bajtów	Maksymalnie 8000 znaków	Typ danych ANSI ze stałą szerokością
nchar(n)	2 – 8000 bajtów	Maksymalnie do 4000 znaków	Typ danych Unicode ze stałą szerokością
varchar(n)	1 – 8000 bajtów	Maksymalnie 8000 znaków	Typ danych ANSI ze zmienną szerokością
varchar(max)	Do 2GB	Do 1 073 741 824 znaków	Typ danych ANSI ze zmienną szerokością
nvarchar(n)	2 – 8000 bajtów	Maksymalnie 4000 znaków	Typ danych Unicode ze zmienną szerokością
nvarchar(max)	Do 2GB	Do 536 870 912 znaków	Typ danych Unicode ze zmienną szerokością
text	Do 2GB	Do 1 073 741 824 znaków	Typ danych ANSI ze zmienną szerokością
ntext	Do 2GB	Do 536 870 912 znaków	Typ danych Unicode ze zmienną szerokością

Ad 6. Binarne typy danych

Wielokrotnie zachodzi potrzeba przechowywania danych binarnych, na przykład zdjęć. SQL server oferuje następujące typy do obsługi tych danych:

Typ danych	Zajmowana pamięć	Przeznaczenie
binary(n)	1 – 8000 bajtów	Przechowuje dane binarne o ustalonym rozmiarze
varbinary(n)	1 – 8000 bajtów	Przechowuje dane binarne o zmiennym rozmiarze
varbinary(max)	Do 2GB	Przechowuje dane binarne o zmiennym rozmiarze
image	Do 2GB	Przechowuje dane binarne o zmiennym rozmiarze

Poznaliśmy już wszystkie potrzebne nam do pracy typy danych. Spróbujmy teraz rozwiązać następujący problem. Chcemy wyświetlić dla każdego produktu nastepujący wiersz wyniku: „Produkt o ID = *<ID produktu>* ma rozmiar *<rozmiar produktu>*”. Z pomocą przychodzi nam funkcja konwersji typu *CAST*:

```
SELECT 'Produkt o ID = ' + CAST(p.ProductID AS varchar(200)) + ' ma rozmiar ' + p.Size AS
Zdanie
FROM SalesLT.Product p
```

Składnia funkcji *CAST* jest następująca: zaraz po poleceniu *CAST* otwieramy nawias, wpisujemy wartość do konwersji, następnie piszemy słowo *AS*, po nim nazwę żądanego typu, zamykamy nawias i gotowe. Istnieje jeszcze polecenie *CONVERT*. Rózni się ono kolejnością parametrów i zamienia słowo kluczowe *AS* na przecinek:

```
SELECT 'Produkt o ID = ' + CONVERT(varchar(200), p.ProductID) + ' ma rozmiar ' + p.Size AS
Zdanie
FROM SalesLT.Product p
```

Tak naprawdę nie są to jedyne różnice. *CONVERT* ma o wiele większe możliwości, można także dodać trzeci parametr polecenia decydujący o formacie konwersji. Generalnie jeśli wsytarcza nam użycie funkcji *CAST*, to nie należy używać *CONVERT*.

Zadania

1. Jakiego typu jest kolumna *Size* w tabeli *Product*?
2. Jaką maksymalną ilość adresów może przechowywać tabela *Address*?
3. Czy baza zapewnia możliwość przechowania produktu o wadze 67.536?
4. Czy baza umożliwia przechowywanie nazw produktów w wielu językach?
5. Czy czas modyfikacji powiązania adresu i klienta można zapamiętać z dokładnością co do sekundy?
6. Dla każdego klienta znajdź najstarszą datę modyfikacji jego powiązania z jakimkolwiek adresem. Wynik powinien zawierać jedną kolumnę, a każdy wiersz powinien wyglądać

naztępująco: „Ostatnia data modyfikacji dla <imię>, <nazwisko> (ID = <identyfikator klienta>) to <data>.”.

7. Dla produktów posiadających wagę wyświetl stosunek ich ceny do ich wagi. Wynik powinien być dokładny do 5 miejsc dziesiętnych.
8. Dla każdego zamówienia wyświetl datę jego dostawy. Data powinna zawierać rok, miesiąc i dzień, a nie powinna zawierać godzin, minut, sekund i milisekund.
9. Wybierz wszystkie czerwone lub żółte produkty, które mają podany rozmiar jako liczbę (np. 52, a nie M). Dla każdego z tych produktów utwórz dodatkową kolumnę zawierającą rozmiar podniesiony do czwartej potęgi.
10. Zmodyfikuj następujące zapytanie

```
select cast('2007-12-12 11:11:11.111' as datetime)
```

tak, aby wypisało tę samą datę, ale z godziną 00:00:00.000. Użyj wyłącznie funkcji konwertujących. Wskazówka jeśli konwertujemy datę na napis za pomocą funkcji *CONVERT*, to powinna ona mieć dodatkowy, trzeci parametr o wartości 120.

Funkcje wbudowane

Najczęściej używane i najbardziej przydatne funkcje, jakie oferuje nam *SQL Server* możemy podzielić na następujące grupy:

1. **Funkcje daty i czasu;**
2. **Funkcje matematyczne;**
3. **Funkcje operujące na napisach;**

Ad 1. Aby dokładniej wyjaśnić operacje na datach wprowadźmy sobie tabelę przedstawiającą składniki daty:

Składnik	Oznaczenie
Rok	Yy, yyyy
Kwartał	Qq, q
Miesiąc	Mm, m
Dzień roku	Dy, y
Dzień	Dd, d
Tydzień	Wk, ww
Dzień tygodnia	Dw, w
Godzina	Hh
Minuta	Mi, n
Sekunda	Ss, s
Milisekunda	ms

Poniższa tabela przedstawia funkcje operujące na dacie i czasie wraz z przykładowym użyciem:

Nazwa i składnia funkcji	Przeznaczenie	Przykład użycia
dateadd(składnik, liczba, data)	Funkcja zwraca datę uzyskaną przez dodanie zadanej liczby	Aby uzyskać dzień przypadający trzy kwartały po dniu 2006-12-09 należy wydać polecenie `SELECT DATEADD(q, 3, '2006-12-09')`

	składników do daty	
datediff(składnik, data początkowa, data końcowa)	Funkcja zwraca różnicę między datą końcową, a początkową wyrażoną w składnikach daty.	Aby dowiedzieć się ile dni jest pomiędzy datą 2007-12-09, a 2001-01-30 należy zapytać: `SELECT DATEDIFF(d, '2001-01-30', '2007-12-09')`
datepart(składnik, data)	Funkcja zwraca żądany składnik z daty	Aby uzyskać numer tygodnia z daty 2005-06-30 należy wydać polecenie: `SELECT DATEPART(wk, '2005-06-30')`
day(data)	Zwraca numer dnia w miesiącu	`SELECT DAY('2008-12-12')`
month(data)	Zwraca numer miesiąca w roku	`SELECT MONTH('2008-11-12')`
year(data)	Zwraca rok	`SELECT YEAR('2008-11-12')`
getdate()	Zwraca bierzącą datę i godzinę	`SELECT GETDATE() AS AktualnyCzas`

We wszystkich powyższych przykładach polecenie *SELECT* zostało użyte w celu wyświetlenia wartości. Należy pamiętać, że każda z powyższych funkcji może być użyta do konstrukcji warunku *WHERE*. Dla przykładu chcielibyśmy poznać te modyfikacje przypisań klientów do adresów, które miały miejsce między 20, a 25 tygodniem roku:

```
SELECT ca.*
FROM SalesLT.CustomerAddress ca
WHERE DATEPART(wk, ca.ModifiedDate) BETWEEN 20 AND 25
```

Ad 2. Funkcje matematyczne zestawimy podobnie jak fukkje daty I czasu:

Nazwa i składnia funkcji	Przeznaczenie	Przykład użycia
ABS(liczba)	Zwraca wartość bezwzględną liczby	`SELECT ABS(-4.234)`
ACOS(liczba rzeczywista)	Arcus cosinius, zwraca kąt w radianach	`SELECT ACOS(0.0)`
ASIN(liczba rzeczywista)	Arcus sinuc, zwraca kąt w radianach	`SELECT ASIN(0.0)`
ATAN(liczba rzeczywista)	Arcus tangens	`SELECT ATAN(1.0)`

CEILING(liczba)	Funkcja sufit; zwraca najmniejszą liczbę całkowitą większą bądź równą podanej	`SELECT CEILING(4.5)`
COS(liczba)	Zwraca cosinus kąta podanego w radianach	`SELECT COS(1.5)`
COT(liczba)	Zwraca cotangens kąta podanego w radianach	`SELECT COT(2.5)`
DEGREES(liczba)	Konweruje radiany na stopnie	`SELECT DEGREES(1.5)`
EXP(liczba)	Zwraca zadaną potęgę liczby **e**	`SELECT EXP(4.0)`
FLOOR(liczba)	Funkcja podłoga; zwraca największą liczbę całkowitą mniejszą bądź równą zadanej	`SELECT FLOOR(-4.2)`
LOG(liczba)	Zwraca logarytm naturalny z zadanej liczby	`SELECT LOG(EXP(1))`
LOG10(liczba)	Zwraca logarytm dziesiętny z zadanej liczby	`SELECT LOG10(100)`
PI()	Zwraca wartość ∏	`SELECT COS(PI())`
POWER(liczba, potega)	Podnosi liczbę do zadanej potęgi	`SELECT POWER(2, 10)`
RADIANS(liczba)	Konwertuje wartość kąta podaną w stopniach na radainy	`SELECT RADIANS(90)`
RAND()	Losuje liczbę rzeczywistą z zakresu od 0 do 1	`SELECT RAND()`
ROUND(liczba, precyzja)	Zaokrągla liczbę do wskazanej precyzji	`SELECT ROUND(4.6436, 2)`
SIGN(liczba)	Zwraca -1 dla liczb ujemych, 0 dla zera i 1 dla liczb dodatnich	`SELECT SIGN(-34), SIGN(0), SIGN(32)`
SIN(liczba)	Wraca wartość funkcji sinus dla kąta zadanego w radianach	`SELECT SIN(1.5)`
SQURT(liczba)	Zwraca pierwiastek kwadratowy z zadanej liczby	`SELECT SQRT(1024)`
SQUARE(liczba)	Podnosi liczbę do 2 potęgi	`SELECT SQUARE(4)`

TAN(liczba)	Zwraca wartość funkcji tangens dla argumentu podanego w radianach	`SELECT TAN(4)`

Ad 3. Teraz przedstawimy spis funkcji operujących na napisach:

Nazwa i składnia funkcji	Przeznaczenie	Przykład użycia
ASCII(napis)	Zwraca numer kodu ASCII pierwszego znaku	`SELECT ASCII('01234')`
CHAR(liczba)	Zwraca znak ASCII odpowiadający zadanej liczbie	`SELECT CHAR(48)`
CHARINDEX(napis1, napis2, liczba)	Zwraca pierwsze wystapiwnie *napisu1* w *napisie2* zaczynając od podanej pozycji	`SELECT CHARINDEX('aba', 'ababbbaaaabaooooaba', 4)`
SOUNDEX(napis)	Wylicza indeks podobieństwa dla napisu	`SELECT SOUNDEX('aba'), SOUNDEX('abab')`
DIFFERENCE(napis1, napis2)	Zwraca podobieństwo 2 napisów; wartość od 0 do 4; im wyższa wartośćtym bardziej podobne napisy	`SELECT DIFFERENCE('aba', 'abab')`
LEFT(napis, x)	Zwraca *x* pierwszych znaków napisu	`SELECT LEFT('567890', 3)`
LEN(napis)	Zwraca długość napisu	`SELECT LEN('567890')`
LOWER(napis)	Zmienia wszystkie litery w napisie na małe	`SELECT LOWER('Jan Kowalski')`
LTRIM(napis)	Usuwa białe znaki z lewej strony napisu	`SELECT LTRIM('   rrr')`
PATINDEX(wzorzec, napis)	Zwraca pierwszą pozycję pierwszego wystapienia wzorca w napisie	`SELECT PATINDEX('%rt%', 'Robert')`

REPLACE(napis, stary fragment, nowy fragment)	Zastępuje w napisie wszystkie wystąpienia starego fragmentu nowym	`SELECT REPLACE('abcdefghicde','cde','xxx')`
REPLICATE(napis, liczba)	Powiela napis zadaną ilością razy	`SELECT REPLICATE('as', 20)`
REVERSE(napis)	Odwraca napis	`SELECT REVERSE('lolek')`
RIGHT(napis, x)	Zwraca *x* ostatnich znaków napisu	`SELECT RIGHT('567890', 3)`
RTRIM(mapis)	Usuwa białe znaki z prawej strony napisu	`SELECT RTRIM('rrr              ') + 'koniec'`
SPACE(liczba)	Zwraca zadaną liczbę spacji	`SELECT 'Jan' + SPACE(1) + 'Kowalski'`
STR(liczba)	Konwertuje liczbę na napis	`SELECT STR(435)`
STUFF(napis1, x, y, napis2)	Usuwa z *napisu1 y* znaków zaczynając od pozycji *x* i wstawia w to miejsce *napisu2*	`SELECT STUFF('abcdef', 2, 3, 'ijklmn')`
SUBSTRING(napis, x, y)	Zwraca napis będący wycinkiem *y* znaków *napisu1* zaczynając od pozycji *x*	`SELECT SUBSTRING('123456789', 2, 4)`
UPPER(napis)	Zwraca napis z samymi dużymi literami	`SELECT UPPER('Jan Kowalski')`

Zadania

1. Napisz zapytanie, które wypisze następujące daty: aktualną, za 1 miesiąc od aktualnej i za 30 dni od aktualnej.
2. Za ile dni będzie Nowy Rok?
3. Dla każdej modyfikacji powiązania klienta z adresem wyświetl w osobnych kolumnach nastepujące dane: rok, miesiąc i dzień.
4. Wylosuj liczbę z zakresu [66, 2322).
5. Wsród fuknkcji matematycznych znajdź funkcje odwrotne i wykaż ich odwrotność przez złożenie.

6. Dla wszystkich klientów utwórz kolumnę zawierająca imię i nazwisko, a następnie za pomocą odpowiednich funkcji wybierz z niej tylko imię.
7. Dla dowolnego modelu każdego produktu wybierz z jego opisu w języku angileskim trzeci wyraz tego opisu.
8. Wybierz dowolny produkt wraz z jego opisem. Następnie wyświetl ten opis lecz bez dwóch ostatnich wyrazów. Powtórz zadanie nie wyświetlając tylko przedostaniego wyrazu.
9. Dla klientów mieszkających w Kanadzie zamień wielkość liter w ich imionach i nazwiskach. Przykład: Jan Kowalski zamień na jAN kOWALSKI.
10. Zbuduj zapytanie, które zwróci tabelę zawierającą 10 wierszy, w której pierwszy zawiera 10 powtórzonych słów sql bez odstępów między nimi, drugi 9, trzeci 8, itd.

Grupowanie, funkcje agregujące i filtry grup

Grupowanie to kolejny element języka SQL, bez którego trudno sobie wyobrazić pisanie zapytań. Na początku tego rozdziału zestawmy sobie tak zwane funkcje agregujące, czyli takie które na podstawie danych z kolumny (lub jej części) wyliczają pojedynczą wartość:

Funkcja	Opis
AVG	Zwraca średnią wartość
COUNT	Zwraca liczbę wierszy
MAX / MIN	Zwraca największą / najmniejszą wartość
SUM	Sumuje
STDEV	Zwraca odchylenie standardowe
VAR	Zwraca wariancję

Użycie tych funkcji na całej tabeli jest bardzo proste. Poniższe zapytanie zwraca ilość produktów oraz ich średnią cenę:

```
SELECT COUNT(*) AS Ilosc, AVG(p.ListPrice) AS SredniaCena
FROM SalesLT.Product p
```

Przejdźmy teraz do zagadnienia badziej skomplikowanego, czyli podziału tabeli na grupy. Wykonajmy na początek proste zapytanie zwracające nazwy produktów wraz z ich kategoriami głównymi i podkategoriami:

```
SELECT pc1.Name AS MainCategory, pc.Name AS Category, p.Name
FROM
        SalesLT.Product p
                INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
                INNER JOIN SalesLT.ProductCategory pc1 ON pc.ParentProductCategoryID =
pc1.ProductCategoryID
```

Jak widzimy istnieje wiele produktów w jednej kategorii głównej. Chcielibyśmy się teraz dowiedzieć ile produktów znajduje się w każdej z kategorii głównych i jaka jest średnia cena w danej kategorii. Przyjmujemy więc, że grupami są kategorie i zadajemy takie pytanie:

```
SELECT pc1.Name AS MainCategory, COUNT(*) AS Count, AVG(p.ListPrice) AS Avg
FROM
        SalesLT.Product p
                INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
                INNER JOIN SalesLT.ProductCategory pc1 ON pc.ParentProductCategoryID =
pc1.ProductCategoryID
GROUP BY pc1.Name
```

Otrzymaliśmy podział na główne grupy wraz z żądanymi ilościami. Teraz rozbijemy tą statystygę na podkategorie:

```sql
SELECT pc1.Name AS MainCategory, pc.Name AS Category, COUNT(*) AS Count, AVG(p.ListPrice) AS
Avg
FROM
        SalesLT.Product p
                INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
                INNER JOIN SalesLT.ProductCategory pc1 ON pc.ParentProductCategoryID =
pc1.ProductCategoryID
GROUP BY pc1.Name, pc.Name
```

Zauważ, że można gruować po wielu kolumnach. Warty spostrzeżenia i zapamiętania jest fakt, że jeżeli używamy grupowania, to w klauzuli *SELECT* dopuszczalne jest tylko wypisanie reprezentantów grup lub podgrup oraz użycie funkcji agregujących na innych kolumnach. Jeśli ten warunek nie zostanie spełniony, to polecenie nie zostanie wykonane i ten błąd należy do dość często popełnianych.

Skomplikujmy sobie nieco zapytanie dodając do powyższych warunków następne:

- Interesują nas wszystkie produkty oprócz czerwonych;
- Wynik chcemy mieć posortowany rosnąco pod względem nazwy głównej kategorii oraz w ramach tej samej kategorii głównej malejąco po ilości przedmiotów w podgrupie;
- Chcemy wyświetlić tylko te podgrupy, w których średnia cena jest wyższa niż średnia cena wszystkich produktów:

```sql
SELECT pc1.Name AS MainCategory, pc.Name AS Category, COUNT(*) AS Count, AVG(p.ListPrice) AS
Avg
FROM
        SalesLT.Product p
                INNER JOIN SalesLT.ProductCategory pc ON p.ProductCategoryID =
pc.ProductCategoryID
                INNER JOIN SalesLT.ProductCategory pc1 ON pc.ParentProductCategoryID =
pc1.ProductCategoryID
WHERE p.Color <> 'Red'
GROUP BY pc1.Name, pc.Name
HAVING AVG(p.ListPrice) > (SELECT AVG(p1.ListPrice) FROM SalesLT.Product p1)
ORDER BY MainCategory, Count DESC
```

Aby założyć warunek na grupę musimy użyć polecenia *HAVING*. Klauzula *WHERE* jest tu niewystarczająca.

Zadania

1. Znajdź ilość adresów w bazie *AdventureWorksLT*.
2. Znajdź wariancję wartości zamówień (*TotalDue*).
3. Ile adresów nie ma wypełnionej drugiej linii (wartość *AddressLine2* jest pusta)?
4. Znajdź średnią wagę produktów z kategorii ‘*Bikes*’.
5. Ile jest klientów w bazie, którzy mają literę ‘g’ w imieniu lub nazwisku. W wyniku powinna pojawić się tylko liczba.
6. Napisz zapytanie zwracające ilość adresów dla każdego z klientów mieszkających w *Stanach Zjednoczonych*. Klienta powinny opisywać dwie kolumny: imię i nazwisko.
7. Dla każdego kraju i miasta w bazie znajdź ilość adresów w tym mieście. Pierwsza kolumna powinna zawierać nazwę państwa, druga – nazwę miasta, a ostatnia ilość adresów.

8. Dla każdego zamówienia znajdź nastepujące informacje: imie i nazwisko zamawiającego oraz cenę najdroższego produktu z tego zamówienia. Interesują nas tylko zamówienia ze *Stanów Zjednoczonych*.
9. Wybierz imię i nazwisko klienta, którego powiązanie z adresem zostało zmodyfikowane najpóźniej.
10. Wybierz imię i nazwisko klienta, którego powiązanie z adresem zostało zmodyfikowane najwcześniej w roku 2002.
11. Wybierz łączną wagę wszystkich zamówionych czerwonych produktów.
12. Dla każdego z produktów w bazie, znajdź informację ile sztuk tego produktu kiedykolwiek zostało zamówione. Jeśli produkt nie był nigdy zamówiony, to w wyniku powinno być zero. Wynik powinien być posortowany malejąco względem ilości zamówionych produktów.
13. Wybierz te podkategorie, w których średnia cena produktów jest wieksza niż 134$.
14. Dla każdego z dostępnych kolorów produktów, znajdź ilość produktów każdego z kolorów.
15. Jaki jest kolor produktów, których najwiecej zamówili mieszkańcy Stanów Zjednoczonych? Wynik powinien zwrócić tylko nazwę koloru.
16. Wybierz te miasta, do których dostarczono przynajmniej 2 rachunki. Wyświetl nazwę miasta i ilość dostarczonych rachunków.
17. Znajdź nazwę miasta, którego mieszkańcy łącznie złożyli zamówienia o największej sumie. W wyniku powinna znaleźć się dodatkowo sumaryczna wartość tych zamówień oraz ich ilość.

Tabele tymczasowe i wyrażenie WITH

W wielu przypadkach pisząc zapytanie SQL możemy stwierdzić, że staje sie ono za bardzo skomplikowane i zaczynamy tracić kontrolę nad tym, co tak naprawdę się w nim dzieje. W przypadku takim chcielibyśmy mieć możliwość rozbić zapytanie na kilka mniejszych korzystających z siebie. Tu z pomocą przychodzą nam tabele tymczasowe. Weźmy przykładowe skonstruowane w rozdziale o podzapytaniach pytanie SQL:

```
SELECT
        a.AddressID, c1.CustomerID, c1.FirstName
FROM
        SalesLT.Address a
                INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
                INNER JOIN
                                (
                                        SELECT c.*
                                        FROM SalesLT.Customer c
                                        WHERE c.FirstName LIKE '%k%') c1 ON ca.CustomerID =
c1.CustomerID
```

Nie jest ono bardzo skomplikowane, ale może nam posłużyć jako przykład w prezentacji mechanizmu tabel tymczasowych. Wykonując poniższe zapytanie przeniesiemy wynik podzapytania do tabeli tymczasowej o nazwie *#c1*:

```
SELECT c.*
INTO #c1
FROM SalesLT.Customer c
WHERE c.FirstName LIKE '%k%'
```

Pobierzmy teraz wszystkie dane z tabeli tymczasowej, aby sprawdzić, czy że rzeczywiście została utworzona:

```
SELECT *
FROM #c1
```

Uwaga! Nazwa lokalnej tabeli tymczasowej musi zaczynać się od znaku *#*, a globalnej (widocznej niezależnie od połączenia) od znaków *##*. Należy również zauważyć, że zapytanie automatycznie utworzy kolumny o odpowiednich nazwach i typach. Wykonajmy teraz zapytanie korzystające z utworzonej tabeli tymczsowej, ktrego wynik jest taki sam jak bardziej skomplikowanego zapytania z podzapytaniem:

```
SELECT
        a.AddressID, c1.CustomerID, c1.FirstName
FROM
        SalesLT.Address a
                INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
                INNER JOIN #c1 c1 ON ca.CustomerID = c1.CustomerID
```

Rozbiliśmy sobie w ten sposób zapytanie na dwa kroki. Należy zauważyć, że polecenie *SELECT ... INTO* nie wykona się drugi raz, ponieważ będzie chciało utworzyć tabelę o nazwie *#c1*, a ta już istnieje. Przed ponownym wykonaniem tego zapytania należy usunąć istniejącą tabelę tymczasową:

```
DROP TABLE #c1
```

Innym sposobem rozbicia zadania na mniejsze kroki jest użycie słowa kluczowego *WITH*. Jego zaletą jest to, że nie trzeba przejmować się czy tabela o danej nazwie już istnieje, czy nie:

```sql
WITH c1 AS
(
	SELECT c.*
	FROM SalesLT.Customer c
	WHERE c.FirstName LIKE '%k%'
)
SELECT
	a.AddressID, c1.CustomerID, c1.FirstName
FROM
	SalesLT.Address a
		INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
		INNER JOIN c1 c1 ON ca.CustomerID = c1.CustomerID
```

Zadania

1. Utwórz kopię tabeli Address używając mechanizmu tabel tymczasowych. Wybierz wszystkie dane z tej kopii.
2. Utwórz tabelę tymczasową zawierającą wszystkich klientów mieszkających w Kanadzie wraz z ilością zamówień, które zostały do nich wysłane.
3. Utwórz dowolne zapytanie do bazy AdventureWorksLT korzystające z polecenia WITH.
4. Utwórz tabelę tymczasową zawierającą nazwy wszystkich produktów wraz z ich kategoriami głównymi. Następnie korzystając z tej tabeli tabeli utwórz inną tabelę zawierającą ilości produktów w każdej kategorii głównych. Napisz nastepnie zapytanie znajdujące karegorię główną, w która zawiera najmniej produktów.
5. Używając podzapytania w klauzuli FROM i nieużywając tabel tymczasowych wykonaj następujące zadanie:

   Wybierz te kategorie główne, które zawierają więcej produktów niebieskich niż czerwonych lub więcej prouktów niebieskich niż żółtych.

   Następnie utwórz alternatywną wersje tego zapytania w następujący sposób. Utwórz tabelę tymczasowoą zawierającą nazwy kategorii głównych i liczbę zawartych w nich produktów o kolorach niebieskim, czerwonym i żółtym. Korzystając z tej tabeli ułóż zapytanie dające wynik opisany w powyższym paragrafie.
6. Zmodyfikuj wynik zadania 3. tak, aby zamast tabeli tymczasowej użyte było polecenie WITH.
7. Używając dwóch tabel tymczasowych i ich złączenia w klauzuli FROM wykonaj następujące zadanie: dla każdego z krajów znajdź liczbę klientów i adresów.
8. Utówrz tabelę tymczasową zawierającą następujące dane dla każdego z produktów: nazwę, cenę, kolor, nazwę kategorii i nazwę podkategorii. Nastepnie bazując na tej tabeli napisz dwa zapytania dające następujące wyniki:
   a. Ilość produktów o kolorze czerwonym w każdej z kategorii;
   b. Dla każdego koloru sprawdź w ilu kategoriach występuje.

   Zauważ, że tylko raz musiałeś/aś dokonywać uciążliwego łączrnia tabel.

Trochę programowania: DECLARE, SET, PRINT, IF oraz WHILE

Do tej pory pisaliśmy wyłącznie zapytania SQL, teraz przystąpmy do napisania swojego pierwszego programu w tym języku. Będzie to bardzo prosty program, który za pomocą polecenia *DECLARE* zadeklaruje zmienną typu całkowitego (nazwa zmiennej jest poprzedzona znakiem @), oraz przypisze do tej zmiennej wartość 56 jak równieżwypisze wspomnianą zmienną na ekranie (uwaga! Należy wykonać wszystkie linie naraz):

```
DECLARE @x int

SET @x = 56

PRINT @x
```

Napiszmy teraz program wykorzystujący instrukcję *IF ... ELSE*. Instrukcja ta sprawdza warunek, i jeśli jest on prawdziwy, to wykonuje instrukcje wypisane po *IF*, a w przeciwnym przypadku te zawarte po *ELSE*. Uwaga! Jeśli po sprawdzeniu warunku ma być wykonana więcej niż jedna linia kodu, to linie te trzeba zamknąć pomiędzy slowami *BEGIN* i *END*:

```
DECLARE @color varchar(200)

SET @color = 'red'

IF @color = 'red'
	PRINT 'Wybierz inny kolor'
ELSE
BEGIN

	SELECT p.*
	INTO #temp
	FROM SalesLT.Product p
	WHERE p.Color = @color

	SELECT *
	FROM #temp

	DROP TABLE #temp

END
```

Wykonajmy powyższy progam, a następnie zmieńmy początkowe przypisanie zmiennej *@color* na *'blue'* i wykonaj ponownie. Instrukcje *IF ... ELSE* można łączyć w tak zwane wyrażenia *ELSE IF*, które mogą sprawdzić kilka warunków, a dopiero na końcu wykonuje kod nie spełniający, żadnego z powyższych warunków:

```
DECLARE @color varchar(200)

SET @color = 'blue'

IF @color = 'red'
	PRINT 'Kolor czerwony'
ELSE IF @color = 'blue'
	PRINT 'Kolor niebieski'
ELSE IF @color = 'yellow'
	PRINT 'Kolor żółty'
ELSE
	Print 'Inny kolor'
```

Spróbujmy zmienić ścieżkę wykonania programu manipulując początkową wartością zmiennej *@color*. Instrukcja *IF* kończy się na jednym „przebiegu”. Linia *ELSE* jes opcjonalna i nie musimy jej używać:

```
DECLARE @color varchar(200)

SET @color = 'blue'

IF @color = 'red'
       PRINT 'Kolor czerwony'
```

Jeśli chcemy te same instrukcje wykonać wiele razy musimy użyć pętli *WHILE*. Napiszmy program, który wypisuje liczby od 1 do 10:

```
DECLARE @x tinyint

SET @x = 1

WHILE @x <= 10
BEGIN

       PRINT @x

       SET @x = @x + 1

END
```

Progam ten na początku inicjalizuje zmienną *@x* wartością 1. Następnie sprawdza czy wartość *@x* jest mniejsza od 10. Ponieważ *@x = 1*, to wypisuje wartość zmiennej *@x* czyli 1, a następnie zwiększa wartość *@x* o 1 i ponownie sprawdza warunek, itd. Wykonanie pętli zakończy się wówczas, gdy zmienna *@x* osiągnie wartość 11, bo warunek podany w lini *WHILE* nie będzie już prawdziwy.

Zadania

1. Zadeklaruj i zainicjuj dwie zmienne: liczbę 5 i napis 'To jest liczba '. Następnie korzystając z tych zmiennych wypisz tekst: 'To jest liczba 5!'.
2. Znajdź liczbę wszystkich przedmiotów. Jeśli jest ich więcej niż 500, to wypisz tekst: 'Przedmiotów jest więcej niż tysiąc!'. W przeciwnym wypadku program nie powinien nic robić.
3. Napisz program, który pobierze do zmiennych ilości przedmiotów czerwonych i niebieskich oraz w zależności od tych ilości wypisze odpowiedni tekst: 'Niebieskich jest więcej niż czerwonych', bądź 'Czerwonych jest więcej niż niebieskich'.
4. Wybierz losową ilość wierszy z wybranej losowo tabeli: Product, Customer lub Address.
5. Wypisz słowo 'Produkt' tyle razy ile jest produktów w bazie.
6. Napisz program, który używając pętli WHILE policzy następującą sumę 3 + 5 + 7 + ... + n, gdzie n jest nieparzyste i określane na początku programu.

Konstrukcja CASE

Aby łatwiej sprawdzić wielokrotny warunek możemy wykorzystać polecenie *CASE*. Spróbujmy jeszcze raz napisać program z kolorami:

```
DECLARE @color varchar(200)

SET @color = 'ed'

PRINT CASE
        WHEN @color = 'red' THEN 'Kolor czerwony'
        WHEN @color = 'blue' THEN 'Kolor niebieski'
        WHEN @color = 'yellow' THEN 'Kolor żółty'
        ELSE 'Inny kolor'
END
```

Zauważmy, że instrukcja *PRINT* pojawia się teraz tylko raz, a wyrażenie *CASE* dostarcza argumentu do wypisania. Nasze polecenie możmy skrócić jeszcze bardziej „wypychając” zmienną color wyżej, do instrujci *CASE* z poszczególnych instrukcji *WHEN*:

```
DECLARE @color varchar(200)

SET @color = 'ed'

PRINT CASE @color
        WHEN 'red' THEN 'Kolor czerwony'
        WHEN 'blue' THEN 'Kolor niebieski'
        WHEN 'yellow' THEN 'Kolor żółty'
        ELSE 'Inny kolor'
END
```

Przy używaniu funkcji *CASE* musimy pamiętać, że każda z możliwości musi zwracać wartość tego samego typu. Skoro więc nawet rozbudowane wyrażenie *CASE* zwraca pojedyńczą wartość, to sprubujmy połączyć tę funkcję z poleceniem *SELECT*:

```
SELECT
        p.ProductID,
        CASE p.Color
                WHEN 'Red' THEN 'Czerwony'
                WHEN 'Blue' THEN 'Niebieski'
                ELSE 'Inny'
        END AS KolorNiebieskiCzyCzerwony
FROM
        SalesLT.Product p
```

Powyższe zapytanie oznacza produkty o kolorach czerwonym i niebieskim, a inne kolory traktuje „zbiorcz”. Istniej także możliwość użycia *CASE* w warunku *WHERE*:

```
SELECT p.*
FROM SalesLT.Product p
WHERE p.ListPrice > CASE p.Color
                        WHEN 'red' THEN 100
                        WHEN 'silver' THEN 200
                        WHEN 'blue' THEN 400
                        ELSE 700
                    END
```

Powyższe zapytanie zwraca informacje o produkcie jeśli kolor jest: czerwony i cena większa od 100; srebrny i cena większa od 200; niebieski i cena większa od 400; albo innego koloru i cena jest większa od 700.

Zadania

1. Zlicz ilość wierszy w tabeli Address i w zależności od tej liczby wypisz tekst:
    a. ‘adresów jest mało’, jeśli ich liczba należy do przedziału [0, 50];
    b. ‘adresów jest w sam raz’, jeśli liczba należy do przedziału [51, 100];
    c. ‘adresów jest dużo’, jeśli liczba jest większa niż 100.
2. Zwróćmy uwagę na niespójność oznaczeń rozmiarów produktów: jedne wyrażane są liczbowo, a inne symbolami takimi jak np. XL. Utwórz zapytanie zwracające nazwy produktów wraz z ich symbolicznymi rozmiarami. Zamień następujące przediały na symbole:
    d. Rozmiary do 37 na S;
    e. Rozmiary od 38 do 45 na M;
    f. Rozmiary większe od 45 na L;
3. Wypisz imiona i nazwiska klientów, którzy (użyj konstrukcji CASE w klauzuli WHERE):
    g. Mieszkają w USA i zamówili więcej niż 2 produkty;
    h. Mieszkają w Wielkiej Brytanii i zamówili więcej niż 3 produkty;
    i. Mieszkają w Kanadzie i nie zamówili żadnego produktu.

Modyfikacja danych – polecenia: INSERT, UPDATE oraz DELETE

Do tej pory zajmowaliśmy się wydobywaniem danych z bazy i poza drobnym epizodem z tabelami tymczasowymi sami danych nie modyfikowaliśmy. Spróbujmy więc nadrobić zaległości. Zaczniemy od wastawiania danch do istniejącej tabeli. Zanim zaczniemy odczytajmy największy identyfikator adresu z tabeli *Address*:

```
SELECT MAX(a.AddressID)
FROM SalesLT.Address a
```

Teraz wiemy, że adresy o ID powyżej 11382 są dodanymi przez nas. Z reguły w tabelach identyfikatory są kolumnami autoinkrementującymi, to znaczy przy wstawianiu kolejnego wiersza do tabeli ID jest zwiększany automatycznie o 1 i do takiej kolumny nic już nie wstawiamy, co więcej nie możemy wstawić. Dodajmy jeden adres:

```
INSERT INTO SalesLT.Address(AddressLine1, City, StateProvince, CountryRegion, PostalCode,
ModifiedDate)
VALUES ('Bema 11', 'Wrocław', 'dolnośląskie' ,'dolnośląskie', '50-050', GETDATE())
```

Powyższa instrukcja wstawia tylko jeden adres. Wstawmy teraz 20 nowych adresów. Wygenerujemy je losując z już istniejących adresów 20 sztuk i zmieniając w nich miasto na wrocław (dla lepszego poglądu polecenie to wykonajmy w dwóch krokach: najperw zaznaczmy samo polecenie *SELECT* i je wykonajmy – to będą wiersze, które później zostaną wstawione, a potem całość):

```
INSERT SalesLT.Address(AddressLine1, City, StateProvince, CountryRegion, PostalCode,
ModifiedDate)
SELECT TOP 20 a.AddressLine1, 'Wroc³aw' AS City, a.StateProvince, a.CountryRegion,
a.PostalCode, GETDATE()
FROM SalesLT.Address a
ORDER BY NEWID()
```

Znajdźmy teraz wiersze, które wstawiliśmy (wiemy, że mają *ID > 11382*):

```
SELECT a.*
FROM SalesLT.Address a
WHERE a.AddressID > 11382
```

Adresów tych jest 21, więc tyle ile faktycznie wstawiliśmy.

Zajmiemy się teraz poleceniem *UPDATE*, które modyfikuje dane zawarte w tabelach. Proste zapytanie, które zmienia nazwę miasta w dodanych przez nas adresach na Poznań i dodatkowo adres na ulicę Kościuszki, wygląda następująco:

```
UPDATE SalesLT.Address
SET City = 'Poznań', AddressLine1 = 'ul. Kościuszki'
WHERE AddressID > 11382
```

Teraz zbudujemy polecenie uaktualniające dane o większym stopniu komplikacji, gdyż użyte zostaną dwie złączone tablice. Należy pamiętać, że w jenym poleceniu możemy jednoczeście nadpisywać kolumny tylko z jednej tabeli. Zróbmy takie zadanie: Dla adresów, które mają w polu *AddressLine2* wartość pustą, wsawmy tam e-mail jednej z osób, która jest powiązana z tym adresem. Zadanie to wykonamy według pewnego schematu, którego powinniśmy używać przy bardziej skomplikowanych *UPDATE*-ach. A więc krok pierwszy to skonstruowanie takiego polecenia *SELECT*, które zawiera pola, które będziemy nadpisywać (w naszym wypadku będą to wartości puste) oraz kolumn użytych do tego nadpisania:

```sql
SELECT
	a.AddressID, a.AddressLine2, c.EmailAddress
FROM
	SalesLT.Address a
		INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
		INNER JOIN SalesLT.Customer c ON ca.CustomerID = c.CustomerID
WHERE
	a.AddressLine2 IS NULL
```

Widzimy teraz pusta kolumnę i kolumnę z której chcemy „przepisać" dane. Zauważmy, że wynika zapytania nie zawiera wszystkich wierszy, a jedynie te które mamy zaktualizować. Drugim i ostatnim krokiem jest zamienienie klauzuli *SELECT* klauzulami *UPDATE* i *SET*, przy czym reszta zapytania poczynając od *FROM* pozostaje bez zmian:

```sql
UPDATE
	SalesLT.Address
SET
	AddressLine2 = c.EmailAddress
FROM
	SalesLT.Address a
		INNER JOIN SalesLT.CustomerAddress ca ON a.AddressID = ca.AddressID
		INNER JOIN SalesLT.Customer c ON ca.CustomerID = c.CustomerID
WHERE
	a.AddressLine2 IS NULL
```

Ostatnim poleceniem modyfikującym dane jest *DELETE*, które usuwa wiersze z tabeli. Używając tego polecenia możemy usunąć wszystkie adresy, które dodaliśmy uprzednio:

```sql
DELETE FROM SalesLT.Address
WHERE AddressID > 11382
```

Przy poleceniach *UPDATE* i *DELETE* należy zwracać szczególną uwagę na definicję warunku *WHERE* ponieważ jeżeli popełnimy tu błąd, to zbyt wiele danych może zostać bezpowrotnie zmodyfikowanych. Na szczęście istnieje prosty sposób, który może służyć jako ubezpieczenie w takich sytuacjach. Opiszemy go w następnym rozdziale.

Zadania

1. Bazując na tabeli *Customer* utwórz tabelę tymczasową #Imiona z imionami klientów. Imiona nie powinny się powtarzać! Następnie za pomocą polecenia *INSERT* dodaj do tej tabeli swoje imię.
2. Do tabeli z imionami dodaj za pomocą jednej instrukcji *T-SQL* wszystkie nazwiska klientów występujących w bazie danych. Nazwiska mogą się powtarzać.
3. Dodaj swoje dane do tabeli *Address* i tabeli *Customer*, a następnie je powiąż.
4. Dla adresów, które nie mają drugiej lini adresu przypisz w tym miejscu (*AddressLine2*) wartość pustego napisu '';
5. Dla niebieskich produktów zwiększ cenę o losową wartość z przedziału [0$, 10$].
6. Stosują taktykę dwóch kroków (najpierw *SELECT*, potem *UPDATE*) zwiększ cenę produktom z ketegorii głównej *'Bikes'* o 20%.
7. Usuń wszystkie produkty, z których zysk (*ListPrice - StandardCost*) nie przekracza 200$.
8. Usuń z bazy wszystkie produkty, których identyfikatory dzielą się przez 12 bez reszty i są koloru czerwonego.
9. Usuń klienta o identyfikatorze 18. Jeśli pojawią się problemy, zbadaj dlaczego one wystąpiły i zaproponuj rozwiązanie jak je rozwiązać.

10. Zauważ, że w przykładowej bazie danych wszystkie zamówienia mają tę samą datę (*OrderDate*). Za pomocą jednej instrukcji *UPDATE* spraw aby każda z dat była różna (dodaj / odejmij od nich losową wartość).
11. Zabezpieczając się przed pomyłką za pomocą transakcji wykonaj następującą operację: wszystkim klientom ze *Stanów Zjednoczonych* zamień kraj na *Kanadę* i odwrotnie wszystkim z *Kanady* zamień kraj na *Stany Zjednoczone*.

Transakcje, czyli prosty sposób unikania pomyłek

Transakcja jest zbiorem poleceń, które są wykonywane niepodzilnie, to znaczy, że albo wszystkie zostaną wykonane albo żadne. Przykład transakcji to przelew pomiędzy kontami bankowymi. Dla nas istotne jest, że w języku T-SQL możemy świadomie rozpocząć transakcję, zakończyć ją, bądź wycofać wszystkie zmiany, które wprowadziły polecenia zawarte w transakcji. Cały schemat uniknięcia błędu polega na rozpoczęciu transakcji, wprowadzeniu zmian i ich sprawdzeniu. Jeśli zmiany są zgodne w naszymi oczekiwaniami, zatwierdzamy transakcję, a w przeciwnym przypadku wycofujemy transakcję, przez co unieważniamy dokonane zmiany. Przećwiczmy ten schemat. Rozpoczynamy transakcję:

```
BEGIN TRANSACTION
```

Nastepnie wykonujemy „omyłkowo” zapytanie uaktualniające zapominając o klauzuli *WHERE*:

```
UPDATE SalesLT.Product
SET Weight = 2.34
```

Po sprawdzeniu okazuje się, że wszystkim produktom zmieniliśmy wagę na 2.34, a chcieliśmy tylko tym czerwonym:

```
SELECT *
FROM SalesLT.Product
```

Wycofujemy więc transakcję, sprawdzamy, że błędne zmiany zniknęły i zaczynamy ją jeszcze raz:

```
ROLLBACK TRANSACTION

SELECT *
FROM SalesLT.Product

BEGIN TRANSACTION
```

Wykonujemy tym razem poprawne polecenie i sprawdzamy ponownie wynik.

```
UPDATE SalesLT.Product
SET Weight = 2.34
WHERE Color = 'Red'
```

Ponieważ wynik jest taki jakiego się spodziewaliśmy, więc zatwierdzamy transakcję:

```
COMMIT TRANSACTION
```

Od tej pory wykonana zmiana jest nieodwracalna.

Zapytania dynamiczne i polecenie EXECUTE

Potrafimy już całkiem dobrze pisać zapytania oraz znamy funkcje operujące na napisach. Powstaje więc pytanie: czy możemy skonstruować napis zawierający poprawne zapytanie, a potem uruchomić zapytanie zawarte w tym napisie? Odpowiedź brzmi: TAK. Podobnie jak w wielu poprzednich przypadkach, zrobimy to w dwóch krokach. Pierwszy z nich, to utworzenie napisu i jego wyświetlenie w celu weryfikacji poprawnośi:

```
DECLARE
	@selectCmd varchar(200), @fromCmd varchar(200),
	@whereCmd varchar(200), @cmd varchar(600)

SET @selectCmd = 'SELECT p.ProductID, p.Name'
SET @fromCmd = 'FROM SalesLT.Product p'
SET @whereCmd = 'WHERE p.Color = ''Red'''

SET @cmd = @selectCmd + char(10) + @fromCmd + char(10) + @whereCmd

PRINT(@cmd)
```

Zauważamy, że wypisane polecenie jest poprawne, więc zamieniamy polecenie wypisujące treść zapyania (*PRINT*), na polecenie wykonujące *EXECUTE*:

```
DECLARE
	@selectCmd varchar(200), @fromCmd varchar(200),
	@whereCmd varchar(200), @cmd varchar(600)

SET @selectCmd = 'SELECT p.ProductID, p.Name'
SET @fromCmd = 'FROM SalesLT.Product p'
SET @whereCmd = 'WHERE p.Color = ''Red'''

SET @cmd = @selectCmd + char(10) + @fromCmd + char(10) + @whereCmd

EXECUTE(@cmd)
```

Uruchamiając powyższy program widzimy wynik zapytania. Rozwiążmy teraz problem, do którego zapytanie dynamiczne lepiej pasuje: chcemy zwrócić tabelę składającą się z 20 wierszy i 121 kolumn ponumerowanych od 1 do 121. Cała tabela powinna zawieraćwartość NULL.

```
DECLARE
	@selectCmd varchar(2000), @fromCmd varchar(200), @cmd varchar(2000),
	@counter int

SET @selectCmd = 'SELECT TOP 20'
SET @counter = 1

WHILE @counter <= 121
BEGIN

	SET @selectCmd = @selectCmd + ' NULL AS [' + CAST(@counter AS varchar(3)) + '],'
	SET @counter = @counter + 1

END

SET @selectCmd = SUBSTRING(@selectCmd, 1, LEN(@selectCmd) - 1)

SET @fromCmd = 'FROM SalesLT.Product p'

SET @cmd = @selectCmd + char(10) + @fromCmd

EXEC(@cmd)
```

Zapytania dynamiczne są bardziej pomocne dopiero w połączeniu z konstrukcjami programistycznymi takimi jak instrukcja *IF* lub pętla *WHLIE*. Możemy też używać skróconej formy polecenia *EXECUTE*, a mianowicie *EXEC*.

Polecenie *EXEC* służy także do uruchamiania procedur składowanych

```
EXEC sp_helptext @objname = 'dbo.ufnGetCustomerInformation'
```

Powyższe polecenie uruchamia systemową procedurę *sp_helptext*, która wyświetla treść obiektu wskazanego jako parametr. W naszym przypadku jest to jedna z fukkcji znajdujących się w bazie *AdventureWorksLT*.

Czego nie robić, czyli używanie kursorów i polecenia UNION

Większość problemów z wydajnością baz danych jest spowodowanych nieodpowiednią architekturą (projektem tabel) albo nadmiernym używaniem kursorów. Kursory są mechanizmami, które przetwarzają tabelę „wiersz po wierszu", a jak wiemy bazy danych powinny operować na zbiorach, czyli całych zawartościach tabel. I tak właśnie robiliśmy uprzednio. Są jednak sytuacje, w których nie mamy innego wyjścia jak użycie kursorów. Przykładowy probelm jest następujący: chcemy stworzyć tabelę z jednym wierszem i jedną kolumną, w której będą wypisane wszystkie możliwe kolory produktów oddzielone przecinkami. Aby rozwiązać ten problem, najpierw zbudujemy tabelę, w której każdy wiersz jest w odrębnym wierszu. Następnie przetworzymy uzyskaną tabelę „wiersz po wierszu" za pomocą kursora i zbudujemy napis, który zawiera wszystkie kolory.

```
DECLARE color CURSOR FOR
        SELECT DISTINCT p.Color
        FROM SalesLT.Product p
        WHERE p.Color IS NOT NULL

DECLARE @color varchar(200), @colors varchar(200)

SET @colors = ''

OPEN color

FETCH next FROM color
INTO @color

WHILE @@FETCH_STATUS = 0
BEGIN

        SET @colors = @colors + @color + ', '

        FETCH next FROM color
        INTO @color

END

CLOSE color
DEALLOCATE color

SELECT SUBSTRING(@colors, 1, LEN(@colors) - 1) AS Color
```

Polecenie *UNION* łączy dwa (lub więcej) zbiorów, a dokładniej tabel zestawiając je jakby jedna nad drugą. Należy pamietać, że łączone tabele muszą mieć taką samą liczbę kolumn i ich typy muszą się zgadzać. Stwórzmy sobie dwie przykładowe tabele i połączmy je:

```
SELECT 1 AS liczba, CAST('jeden' AS varchar(20)) AS slownie
INTO #T1
INSERT INTO #T1 VALUES(2, 'dwa')
INSERT INTO #T1 VALUES(3, 'trzy')
INSERT INTO #T1 VALUES(4, 'cztery')
INSERT INTO #T1 VALUES(5, 'pięć')


SELECT 2 AS liczba, CAST('dwa' AS varchar(20)) AS slownie
INTO #T2
INSERT INTO #T2 VALUES(4, 'cztery')
```

Następnie połączmy te tabele:

```
SELECT *
FROM #T1

UNION

SELECT*
```

```sql
FROM #T2
```

Zauważmy, że jeśli wiersz występował w dwóch łączonych tabelach, to w wyniku znajduje się tylko raz. Aby wyłączyć automatyczne usuwanie zdublowanych wierszy należy wydać polecenie:

```sql
SELECT *
FROM #T1

UNION ALL

SELECT*
FROM #T2
```

W przypadku kursorów sprawa jest prosta: jeśli możliwe jest zrobienie czegośc bez użycia kursora, to tak właśnie należy postępować. Natomiast polecenie *UNION* ma tyle samo zwolenników, co przeciwników. Niektórzy twierdzą, że jeśli trzeba w zapytaniach używać polecenia *UNION*, to baza jest źle zaprojektowana. Z drugiej strony w nowych wersjach *SQL Server*-a dodawane są polecenia z natury bardzo podobne do *UNION*. Te dwa nowe polecenia to *EXCEPT*, który zwraca różnicę zbiorów:

```sql
SELECT *
FROM #T1

EXCEPT

SELECT*
FROM #T2
```

Oraz *INTERSECT*, który zwraca przecięcie zbiorów:

```sql
SELECT *
FROM #T1

INTERSECT

SELECT*
FROM #T2
```

Zadania

1. Utwórz dwie jednokolumnowe tabele tymczasowe: jedna zawierająca identyfikatory produktów czerwonych, a druga zawierająca identyfikatory produktów o rozmiarze L lub XL. Następnie bazując na tych zbiorach:
    j. wybierz czerwone produkty o kolorze niebieskim i rozmiarach L lub XL dopuszczając powtórzenia;
    k. wybierz czerwone produkty o kolorze niebieskim i rozmiarach L lub XL tak, aby nie wyswietlać żadnego identyfikatora podwójnie;
    l. wybierz produkty, które są czerwone i mają rozmiar L lub XL;
    m. wybierz produkty, które są rozmiaru L lub XL, ale nie są czerwone;
2. Napisz kursor, który dla każdej kategorii wykona zapytanie wybierające listę jej podkategorii.
3. Utwórz tabelę zawierającą jedną komórkę (1 wiersz i 1 kolumnę), w której będą wypisane imiona wszystkich klientów, których imiona zaczynają się na litery od ‘D’ do ‘M’.
4. Przeprowadź eksperyment. Na dwa sposoby (1 UPDATE i kursor) rozwiąż następujący problem. Zwiększ każdemu czerwonemu produktowi cenę o 0.01$. Porównaj czasy działania obu sposobów. Dla uwypuklenia różnic wykonaj oba zapytania po 10 i po 100 razy (Wskazówka: jak automatycznie wykonać zapytanie 100 razy?).

Obracanie tabeli – operator PIVOT

Zastosowanie operatora PIVOT jest proste i służy do obracania tabeli. W instrukcji SELECT należy określić wartości, które mają zostać obrócone. Następujący przykład wykorzystujący bazę danych AdventureWorksLT stosuje w roli kolumn kolejne lata (wyliczane za pomocą funkcji DatePart). Klauzula FROM wygląda całkiem zwyczajnie, poza instrukcją PIVOT. Ta instrukcja tworzy wartość, która ma zostać wyświetlona w wierszach nowotworzonych kolumn. Niniejszy przykład wykorzystuje agregację SUM kolumny TotalDue (pole wyliczane w klauzuli FROM). Następnie operator FOR zostaje użyty do wymienienia wartości koulmny OrderYear, które mają zostać obrócone:

```
SELECT CustomerID, [2001] AS Y2001, [2002] AS Y2002, [2003] AS Y2003, [2004] AS Y2004
FROM
(
	SELECT CustomerID, DATEPART(yyyy, OrderDate) AS OrderYear, TotalDue
	FROM SalesLT.SalesOrderHeader soh
) piv
PIVOT
(
	SUM(TotalDue)
	FOR OrderYear IN ([2001], [2002], [2003], [2004])
) AS child
ORDER BY CustomerID
```