**SQL Server - wprowadzenie**

**1. Wprowadzenie**

**Narzędzia**:

* SQL Server Management Studio: narzędzie do tworzenia zapytań i administrowaniem SQL Serverem;
* Reporting Services Configuration Manager: służy do konfigurowania reporting services;
* SQL Server Configuration Manager: uruchamianie, zatrzymywanie, zmiana parametrów pracy SQL Servera;

**SQL Server Configuration Manager** → SQL Server Services: podgląd z jakich usług SQL Server się składa. Powinieneś uruchomić SQL Server Agent, odpowiedzialny za wykonywanie zaplanowanych działań. Prawy, properties, Service, Start Mode = automatic. Znowu prawym na Agent i Start - trzeba go uruchomić ręcznie.

**Odtwarzanie bazy danych**.

**Końcówki plików**:

* Full Database Backup → gotowa baza danych do ściągnięcia;
* Script → plik tekstowy który po uruchomieniu daje powyższe, czyli gotową bazę danych;
* Warehouse script (DW) → plik do ćwiczenia hurtowni baz danych;
* LT → oznacza Light, czyli uproszczona baza danych Adventure Works. Normalna baza danych Adventure Works jest bardzo złożona;
* .bak → plik kopii bazy danych, która jest do odtworzenia w SQL;
* Samples → przykładowe skrypty;

**Pracuję na pliku bak**.

**Odtwarzanie bazy danych**: SQL musi mieć uprawnienia do odczytu danego pliku - oznacza to, że musi on być w określonym miejscu. Raczej nie ma uprawnień do odczytu pliku z Pobranych. Umieść go w temp. Zaloguj się do SQL Server Management. Prawy na Databases; restore database. Zaznacz Device; wskaż plik. Możesz zmienić nazwę w Database. O opcjach odtwarzania (lewy) możesz okreslić położenie odtwarzanych plików (Restore As).

**Opcje**: Script → New Query Editor Window. Dzięki temu wszystkie wyklikane opcje pojawią się w formie skryptu dzięki czemu ma się kontrolę nad operacją odtwarzania.

**Nomenklatura**.

**Przykład**.

USE [master]

RESTORE DATABASE [AdventureWorks2016] FROM  DISK = N'C:\temp\AdventureWorks2016.bak' WITH  FILE = 1,  MOVE N'AdventureWorks2016\_Data' TO N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\AdventureWorks2016\_Data.mdf',  MOVE N'AdventureWorks2016\_Log' TO N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\AdventureWorks2016\_Log.ldf',  NOUNLOAD,  STATS = 5

GO

**RESTORE DATABASE [AdventureWorks]** → odtwórz bazę danych [jaką?].

**FROM DISK = *ścieżka dostępu do pliku*** → skąd ją odtworzyć.

**WITH FILE = 1** → odtwarzanie którego backupu (pierwszego). W tym przykładzie jest tylko jeden, ale back-upów może być więcej.

**MOVE** *ścieżka dostępu* **TO** *ścieżka dostępu* → skąd dokąd mają zostać przeniesione poszczególne elementy bazy danych.

**NOUNLOAD** → zaszłość z czasów przenoszenia backupów na taśmę. Polecenie mówi żeby nie wyładować taśmy z nośnika.

**STATS** = 5 → informowanie o tym co się dzieje ma się odbywać co 5% tego procesu.

**Potwierdzanie skryptu** → execute lub F5. Każda z komend wykonywanych za pomocą skryptu nie powoduje, że drzewko po lewej stornie się zaktualizuje. Trzeba to zrobić samemu, prawy na Databases i Refresh.

**2. SQL Server Management Studio**

**Szybkie szukanie ikony**: skrót SSMS.

**Logowanie do bazy danych**: podajesz nazwę serwera oraz musisz przejść przez zabezpieczenia. Jeśli jest to tylko Windows Authentication, to hasło i login nie są potrzebne - jest to podstawowe zabezpieczenie Windowsa. Jeśli jednak baza jest ohasłowana, wybierz SQL Server Authentication i podaj login i hasło.

**Węzły** → lista folderów się z nich składa. Folder to węzeł.

**Pierwszy węzeł** → pokazuje wersję serwera oraz jego administratora. Prawym przyciskiem można uruchomić, zatrzymać lub zrestartować jego działanie.

**Properties** (prawym na pierwszy węzeł) → można m.in. ustalić ile pamięci może używać SQL Server (Memory), można zmienić ohasłowanie (Security → Server Authentication) czy lokalizację zapisu pliku bazy danych czy back-upu (Database settings).

**Ustawianie maksymalnej i minimalnej ilości wykorzystywanej przez serwer pamięci** → Propeties, memory.

**Przetłumacz wprowadzone zmiany** → na język Transact SQL, poprzez naciśnięcie przycisku Script. Skrypt pojawia się w okienku skryptu.

**Przykład**.

USE [master]

GO

EXEC xp\_instance\_regwrite N'HKEY\_LOCAL\_MACHINE', N'Software\Microsoft\MSSQLServer\MSSQLServer', N'LoginMode', REG\_DWORD, 2

GO

**USE [master]** → polecenie to mówi, żeby przejść do bazy systemowej master. Przechodzenie do baz danych odbywa się za pomocą polecenia USE lub wybranie odpowiedniej pozycji z menu rozwijanego w lewym górnym rogu.

**GO** → instrukcja mówi, że wcześniejsze polecenie ma zostać wykonane, wysłane do serwera.

**EXEC** → instrukcja mówi, że na serwerze ma się odbyć pewna składowana procedura. Procedura w przykładzie to modyfikacja klucza rejestru konfigurującego ten SQL Server.

**F5** → uruchamianie skryptu w całości lub części. Uruchomienie skryptu w części odbywa się poprzez zaznaczenie interesującego cię fragmentu skryptu.

**Restart SQL Server** → Do dokończenia powyższej konfiguracji należy zresetować SQL Server, prawym na pierwszy węzeł i Restart.

**Pozostałe węzły** → pozostałe węzły, bazy danych (np. Adventure Works) również można osobno podejrzeć (ich właściwości) i zmienić lub sprawdzić np. gdzie są przechowywane ich pliki (prawym przyciskiem na węzeł).

**Sprawdzanie jakie loginy istnieją na serwerze** → węzeł Logins w Security.

**Tworzenie nowego loginu SQL Server** → węzeł Security, Logins, prawym i new login. Login name: leszek. Password: haslo. “Enforce password policy”, odznaczone, co oznacza, że nie wymaga się np. żeby hasło miało specyficzne znaki lub było aktualizowane przy każdym logowaniu. Standardowo zamień zmiany na skrypt i F5. Login zostaje dodany. Aby zalogować się za pomocą nowego loginu należy skorzystać z Connect (ramka Object Explorer, gdzie są węzły), Database Engine i wpisać dane.

Np.

USE [master]

GO

CREATE LOGIN [leszek] WITH PASSWORD=N'haslo', DEFAULT\_DATABASE=[master], CHECK\_EXPIRATION=OFF, CHECK\_POLICY=OFF

GO

– WITH PASSWORD, dodawanie hasła.

**Nadaj loginowi uprawnienia do dostępu do Adventure Works** → być może istnieją lepsze rozwiązania, ale jednym z nich jest nadanie nowemu loginowi status SysAdmin we właściwościach. Pamiętaj tylko, że robisz to z poziomu dotychczasowego loginu Admina, tego docelowego. To tam zmieniasz takie rzeczy w logach.

**Komentarze w SQL** → za pomocą symboli –, czyli dwa myślniki.

**Usuwanie loginu** → prawym na login i delete.

Np.

USE [master]

GO

/\*\*\*\*\*\* Object:  Login [leszek] Script Date: 05.10.2022 20:15:43 \*\*\*\*\*\*/

DROP LOGIN [leszek]

GO

**Odłączenie od serwera** → Disconnect (Object Explorer).

**Management** → węzeł interesujący głównie Administratorów Baz. Znajdują się tam logi i tam właśnie pojawiają się błędy. Management, SQL Server Logs.

**Pliki ErrorLog** → znajdują się one w węźle Management, SQL Server Logs. Są tam logi odnośnie błędów, które pojawiały się podczas korzystania z SQL Servera.

**Otwarcie nowego okna skryptów**: New Query lub Crtl+N.

**USE**: polecenie wyznaczające bazę danych na której chcę pracować. Pamiętaj, że aby uruchomić skrypt zaznacz to pole które chcesz uruchomić i F5.

Np.

USE AdventureWorks

**SELECT**: polecenie wyświetlające dane z tabeli bazy danych. Podaj nazwę tabeli. Gwiazdka oznacza, że wybieram wszystko.

Np.

USE AdventureWorks

SELECT \* FROM Person.Person

**Wielkość znaków w skrypcie**: zazwyczaj nie ma znaczenia, jednak są i takie bazy danych, które konfigurowane są aby rozpoznawać różne wielkości znaków.

**WHERE**: polecenie pozwalające od wyfiltrowanie części rekordów. W prawym dolnym rogu okna skryptu pojawi się informacja ile rekordów (rows) zostało zwróconych. Wyniki zwracane są w postaci tabelki tylko do odczytu. Poniższy przykład pokazuje, że z bazy danych AdventureWorks wybierane są z tabeli Person.Person osoby, których pierwsze imię to “Lee”. Oznacza to, że działa tu wzór: WHERE nazwa kolumny = wartość.

Np.

USE AdventureWorks

SELECT \* FROM Person.Person WHERE FirstName = ‘Lee’

**Zakładka Results i Messages**: Zakładki okna skryptów pokazujące wyniki oraz dodatkowe informacje na temat tego jak wykonywane było dane polecenie. Zakładki znajdują się tam, gdzie wyświetlane są dane.

**Wyniki skryptu w postaci tekstu**: jeśli chcesz zobaczyć wyniki skryptu w postaci tekstu wybierz Results to text lub Crtl+T po czym uruchom skrypt. Pamiętaj, że najpierw skrót klawiszowy, a potem uruchomienie skryptu.

**Wyniki skryptu w postaci tabeli**: jeśli chcesz zobaczyć wyniki skryptu w postaci tabeli wybierz Results to grid lub Crtl+D po czym uruchom skrypt. Pamiętaj, że najpierw skrót klawiszowy, a potem uruchomienie skryptu.

**Wyniki skryptu w postaci pliku**: jeśli chcesz zobaczyć wyniki skryptu w postaci pliku, czyli umieścić je w pliku wybierz Results to file lub Crtl+Shtf+F po czym uruchom skrypt. Pamiętaj, że najpierw skrót klawiszowy, a potem uruchomienie skryptu.

**Wybór 1000 pierwszych rekordów (wierszy)** → Prawym na tabelę; Select Top 1000 Rows. Za pomocą tego kodu, możesz łatwo podmienić 1000 na coś innego, interesującą cię liczbę x pierwszych rekordów. Wzór kodu: SELECT TOP (ileś) [Kolumna1], [Kolumna2], [Kolumnax] FROM [Baza danych].[Schemat].[Nazwa tabeli]

Np.

SELECT TOP (1000) [BusinessEntityID]

      ,[NationalIDNumber]

      ,[LoginID]

      ,[OrganizationNode]

      ,[OrganizationLevel]

      ,[JobTitle]

      ,[BirthDate]

      ,[MaritalStatus]

      ,[Gender]

      ,[HireDate]

      ,[SalariedFlag]

      ,[VacationHours]

      ,[SickLeaveHours]

      ,[CurrentFlag]

      ,[rowguid]

      ,[ModifiedDate]

  FROM [AdventureWorks].[HumanResources].[Employee]

**Modyfikacja randomowych 200 rekordów lub dodanie rekordu** → prawym na tabelę i Edit Top 200 Rows i wprowadzając na dole (Null) nowy rekord musisz jeszcze to potwierdzić za pomocą opcji Execute SQL (Crtl+R). Tutaj nie pokazuje kodu, tylko funkcjonalność.

**Potwierdzenie działania** → Execute SQL (Crtl+R).

**Usuwanie rekordu** → prawym i Delete, ale zaznaczając cały wiersz.

**Zeskryptowanie** → utworzenie nowej tabeli (tabela jest obiektem) za pomocą już istniejącej (skopiowanie skryptu). Wybór istniejącego skryptu: prawy na tabelę, Script Table As, Create to, New query.

**INSERT INTO →** Zeskryptowanie wstawiania rekordu to istniejące tabeli. Prawy na tabelę, script table as, insert to, new query. Pojawi się skrypt. Można ręcznie wprowadzić rekord lub otworzyć edytor (Specify Values for Template Parameters albo Crtl+Shift+M) i określić… Pamiętaj, że dodając rekordy musisz korzystać z apostrofów, inaczej pojawi się błąd “Invalid column name”. Pamiętaj, że prawdopodobnie nie będziesz mógł z palca podać ID, to raczej będzie trzeba pozostawić puste.

Np. (Tabela Human Resources Shift)

– Pojedynczy apostrof: dla tekstów jest ogranicznikiem tekstu. Dla daty/czasu również uzyj apostrofu.

Name: ‘Nazwa’

StartTime: ‘0:00’

Endtime: ‘23:59’

– Kiedy został wprowadzony rekord: defalut, co oznacza, że jeśli istnieje wartość domyślna to ma być do tego pola wstawiona. W tym przypadku oznacza to, że data wstawienia (zmodyfikowania czyli ModifiedDate) będzie domyślna, czyli program sam poda kiedy wprowadziłem rekord.

ModifiedDate: default

Kod wygląda następująco:

USE [AdventureWorks]

GO

INSERT INTO [HumanResources].[Shift]

           ([Name]

           ,[StartTime]

           ,[EndTime]

           ,[ModifiedDate])

     VALUES

           (Zmiana

           ,'0:00'

           ,'23:00'

           ,default)

Go

F5, wprowadź rekord. Żeby powyższe zadziałało należy najpierw zeskryptować Create to, a potem wstawić rekord Insert to.

**Polecenie SELECT** → wylicza kolumny, które mają być zwrócone przez zapytanie. To właściwie to samo co pisanie kodu SELECT \* FROM WHERE…, ale chyba po prostu szybsze. Prawy, script table as, select to, new query.

**Widoki** → zapisane zapytania do bazy danych. Węzeł Views.

**Procedury i funkcje** → węzeł Programmability, Stored procedures. Jest to kod, który będzie uruchamiany po stronie serwera.

**Security** → węzeł Security, który np. posiada dane w Users odnośnie tego kto może wejść do tej bazy danych i co w niej zrobić. Ten węzeł znajduje się zarówno na poziomie bazy danych jak i na poziomie SQL Server.

**Default** → jest wartością, którą wpiszesz w miejscu ModifiedDate gry chcesz żeby SQL wygenerował aktualną datę.

Np.

INSERT INTO [Production].[ScrapReason]

        ([Name]

        ,[ModifiedDate])

  VALUES

        ('Nie ten kolor'

        ,default)

**Znak cytu** → w SQL żeby określić, że obiekt jest tekstem posługujesz się pojedynczym apostrofem, nie podwójnym.

**3. Utrzymywanie skryptów. Solutions & Projects**

**Skrypty bazodanowe** → przechowywane są często w folderach, gdzie trudno jest odnaleźć potrzebny skrypt. W SQL Server można przygotować rozwiązania składające się z wielu projektów.

**Rozwiązanie** → rozwiązanie SQL Server Management Studio to zbiór co najmniej jednego powiązanego projektu. Plik będący rozwiązaniem ma końcówkę .ssmssln.

**Projekty** → to zbiorniki, których programiści używają do organizowania powiązanych plików, takich jak zestawy powszechnie używanych skryptów administracyjnych. Projekty mają końcówkę .ssmssqlproj.

**Repozytorium projektów** → przygotowanie. File, new, project. Wybierz SSMSEmptySqlProject (w twojej wersji może to być SQL Server Scripts). Nadaj nazwę projektowi, wciąż w tym samym oknie, na dole (Name, np. “HR”). Rozwiązanie (solution) też nazwij bardziej ogólnie niż ogólnie (Solution name, np. “AdventureWorks”). Jeśli okno Solution Explorer się nie pokazało, szukaj go w View lub Crtl+Alt+L. Z czego składa się projekt:

* Folder Connections → posiada połączenia między wieloma serwerami, np. testowymi i zadaniowymi. Można tu definiować połączenia między nimi. New Connection i dodaj serwer. Jeśli chcesz utworzyć dla połączenia nowe zapytanie, to prawym na połączenie i New Query. Dodaj jakiś gotowy, przeklej.
* Folder Query → tu pojawią się nowe skrypty. Możesz nadać im nowe nazwy. Możesz dodawać skrypty poza w/w drogą poprzez klik na nazwę projektu, Add, Use existing item.
* Folder Miscellaneous → przechowuje pliki dodatkowe, np. plik z danymi, które trzeba zaimportować do serwera.

**Pojedyncza solucja (rozwiązanie)** → może składać się z wielu projektów. Plik będący rozwiązaniem na końcówkę .ssmssln. Projekty mają końcówkę .ssmssqlproj.

**4. Wprowadzenie do baz relacyjnych**

**Baza danych składa się z tabel** → poszczególne kolumny wyróżniają pojedyncze informacje. Ważne, żeby jedną informację trzymać w jednej kolumnie. Nie mieszać w kolumnie wielu informacji.

**Wybór poszczególnych kolumn tabeli, na przykładzie**. Wzór: Polecenie/a USE i SELECT tytuły kolumn po przecinkach bez przecinka na końcu FROM nazwa tabeli

Np.

USE AdventureWorks

SELECT

p. Name

,p.ProductID

,p.ProductSubcategoryID

FROM

Production.Product p

**JOIN\_wstęp →** Łączenie tabel. Powyższy skrypt oddaje informacje z tych tabel, jednak w ProductSubcategoryID są tylko numery przypisane do podkategorii produktów. Jakie to produkty jest ukryte w innej tabeli, należy zatem skryptem JOIN (o którym później) połączyć te dwie tabele. Tu łączymy tabelę Product z tabelą Product Subcategory (która zawiera nazwę podkategorii, jej identyfikator, identyfikator kategorii do której należy ta podkategoria.

Np.

USE AdventureWorks

SELECT

p. Name

,p.ProductID

,p.ProductSubcategoryID

,s.Name

,s.ProductCategoryID

FROM

Production.Product p

JOIN Production.ProductSubcategory s ON s.ProductSubcategoryID = p.ProductSubcategoryID

**Kolejny przykład pokazuje dodanie nazwy kategorii produktu**. Dodaje linię kodu, który dodaje do tabeli kategorię produktu.

USE AdventureWorks

SELECT

p. Name

,p.ProductID

,p.ProductSubcategoryID

,s.Name

,s.ProductCategoryID

,c.Name

FROM

Production.Product p

JOIN Production.ProductSubcategory s ON s.ProductSubcategoryID = p.ProductSubcategoryID

JOIN Production.ProductCategory c on c.ProductCategoryID = s.ProductCategoryID

**Szybkie sprawdzenie nazw kolumn** → otwórz węzeł tabeli i folder “Columns”.

**SET** → polecenie SET jest używane z UPDATE do określenia, które kolumny i wartości powinny zostać zaktualizowane w tabeli. Wzór: SET kolumna = nowa wartość wartość (tak określisz kolumnę i nową wartość. Pozostałe potrzebne informacje zawarte są w WHERE skryptu UPDATE).

Np.

USE AdventureWorks

SELECT \* FROM Purchasing.PurchaseOrderDetail

UPDATE Purchasing.PurchaseOrderDetail

SET OrderQty = 4

WHERE PurchaseOrderId=1 and PurchaseOrderDetailID =1

SELECT \* FROM Purchasing.PurchaseOrderDetail

**Dobra praktyka relacyjnej bazy danych** → jeśli są jakieś wartości, które da się wyliczyć w oparciu o inne, znajdujące się w tabeli wartości to nie należy przechowywać tej wartości jako kolumny na dysku, ale wyliczać ją w locie. Jeśli da się coś wyliczać, to należy to zrobić, a nie przechowywać na dysku. Na przykładzie aktualizacji rekordu (o aktualizacji później) za pomocą UPDATE.

Np.

USE AdventureWorks

SELECT \* FROM Purchasing.PurchaseOrderDetail

UPDATE Purchasing.PurchaseOrderDetail

SET OrderQty = 4

WHERE PurchaseOrderId=1 and PurchaseOrderDetailID =1

SELECT \* FROM Purchasing.PurchaseOrderDetail

**Diagram bazy danych** →  pomocny gdy mam do czynienia z dużą ilością danych. Należy najpierw ustawić właściciela bazy danych. AdventureWorks, properties, files, owner - wpisz nazwę konta administratora serwera. Wygeneruj skrypt polecenia. Diagramy należy tworzyć na początku pracy z bazą danych, żeby poznać relacje między tabelami. Poniżej:

Np.

USE [AdventureWorks]

GO

EXEC dbo.sp\_changedbowner @loginame = N'sa', @map = false

GO

**Dodawanie diagramu** → Baza danych (AdventureWorks), prawym na Database Diagrams, dodaj tabele według których ma powstać diagram (w przykładzie wszystkie, które pochodzą ze schematu Production - tak mają w nawiasie), Add. Zamknij okno jak wszystko się pojawi. Na przykładzie widać, że kluczową tabelą jest tabela Product. Do niej odwołują się wszystkie tabele w tym schemacie. Relacje to słupki łączące. W szczególności relacjami są połączone ProductCategory, ProductSubcategory i Product. Możesz podejrzeć specyfikacje połączenia.

Widzisz tam co jest kluczem podstawowym (Tabels And Columns Specification, Primary/Unique Key Columns). Klucz podstawowy pozwala na jednoznaczną identyfikację rekordu w tabeli. Dobre tabele takie klucze będą posiadać.

Do tej tabeli przykładowej odwołuje się tabela ProductSubcatogory, która to informację o przynależności do danej kategorii zapamiętuje przy użyciu klucza obcego (Foreign Key Base Table) zapisanego w ProductCategoryID. Czyli jeśli podkategoria produktu była przypisana do kategorii numer 10 to znaczy, że oczekuję, że w ProductCategory znajduje się rekord z identyfikatorem kategorii równym 10 i dzięki tej relacji można tworzyć zapytania, np. JOIN.

**5. Wprowadzenie do pomocy (help)**

**Left** → funkcja wyciągająca określona liczbę znaków z wybranych komórek. Podaj liczbę znaków w drugim argumencie. Wyciągnie wszystko z określonej kolumny. Jeśli chcesz wyciągnąć określoną komórkę dodaj WHERE z np. numerem ID.

Np.

USE Adventureworks

GO

SELECT LEFT(FirstName,1) FROM Person.Person

GO

**Każda funkcja ma dokumentację Internetową** → naciśnij F1 przy wyborze funkcji, a pokazane zostanie jak ona działa, jakie ma argumenty, itp. Help w wersji offline: Help, Manage Help Settings. Za pomocą help możesz dowiedzieć się co oznaczają dane parametry funkcji. Wartości w nawiasach kwadratowych są opcjonalne.

**6. Server Configuration Manager**

**SQL Server Services** → miejsce w którym Manager (SQL Server Configuration Manager) przechowuje wszystkie usługi wchodzące w skład SQL Serwera: SQL Server (silnik bazy danych), SQL Server Agent (usługa pozwala na automatyzację czynności w SQL Server), SQL Server Integration Services (usługa pozwala na import i eksport baz danych w środowisku SQL Server), SQL Server Browser (odpowiada na pytania przychodzące od klientów, którzy nie znają portu na którym pracuje SQL Server).

**Oznaczenia** → zielony trójkąt oznacza, że usługa działa, czerwony kwadrat, że jest zatrzymana.

**Konfiguracja** → prawy przycisk. Właściwości:

* Log on: na jakim koncie pracuje SQL Server, można tu zmienić log i hasło do serwera.
* Service: czy SQL Server ma być uruchamiany przy starcie komputera. Disabled Start Mode użyj gdy chcesz odinstalować SQL Server. Zamiast tego go wyłącz. Ktoś inny może chcieć z niego korzystać.
* Filestream: pozwala na przechowywanie w tabelach SQL Sever informacji zawartej w plikach znajdujących się na systemie operacyjnym.
* AlwaysOn High Availability: pozwala na włączenie opcji AlwaysOn High Availability
* Startup parameters: przy starcie SQL Servera w oparciu o określone parametry startowe dowiaduje się gdzie znajdują się: d (master, pierwsza konfiguracyjna baza danych - każda baza danych składa się z pliku danych i pliku loga), l (wskazuje na lokalizację pliku loga dla bazy master), e (errorlog, pokazuje błędy i patrzysz tam gdy SQL Server nie uruchamia się)
* Advanced: inne, zaawansowane opcje

**Zmiany w konfiguracji w przystawce** → zmiany w konfiguracji usług SQL należy wykonywać w SQL Configuration Manager, a nie w przystawce services.msc, bo w przypadku wykonania zmian w przystawce services.msc można utracić dostęp do zaszyfrowanych danych.

**Zatrzymanie SQL Server** → Stop na SQL Server w Managerze.

**Zmiana uruchomienia usługi na automatyczną** → wcześniej mogła być manual (manualna), czyli ręcznie za każdym razem trzeba ją włączyć. Prawy, właściwości, service, Start mode: automatic. Po czym musisz jeszcze ręcznie ją włączyć naciskając prawym i start.

**Podanie instancji SQL Servera w celu uzyskania odpowiedzi** → nie znasz dokładnej nazwy serwera. Uruchom SQL Management Studio, w momencie logowania, Server name: Browse for more, (musi być włączony SQL Server Browser), Network Servers, szukanie instancji SQL, wybierz jeden ze zgłoszonych serwerów. Jeśli środowisko ma być dobrze zabezpieczone to SQL Server Browser powinien być wyłączony - jeśli to środowisko deweloperskie i ma być wygodniej, a nie bezpieczniej to Browser może być włączony.

**System 32 a 64 bit, różnica** →Programy operacyjne od Windows występują w dwóch różnych wersjach - jako systemy operacyjne 32 i 64-bitowe. Różnica w ilości bitów przekłada się znacząco na funkcjonowanie systemu, dlatego warto znać różnice pomiędzy systemami i wybrać odpowiedni do swoich potrzeb. Ilość bitów określa sposób, w jaki procesor przetwarza informacje. Procesory 32-bitowe są w stanie obsłużyć 4 GB pamięci, natomiast 64-bitowe mogą obsługiwać aż 192 GB pamięci. W architekturze 32- bitowej żaden proces nie może zajmować więcej niż 2 GB, w przypadku 64-bitowych systemów procesy mogą być znacznie większe

**Bity w SQL Server** → SQL Server to program 64-bitowy, ale klienci łączący się z SQL Serverem mogą być 32-bitowi. Po lewej stronie Managera można skonfigurować protokoły sieciowe (SQL Server Network Configuration). Server jest 64-bitowy więc zakładka 32bit jest pusta. Konfiguracja dla serwera 64-bitowego jest w SQL Server Network Configuration. Są tu protokoły - najpopularniejszy to TCP/IP. Named Pipes jest wycofywany, nie należy go konfigurować. Sharp memory pozwala na podłączenie się do serwera z lokalnego komputera.

**Nasłuchiwanie kart sieciowych przez SQL Server** → w TCP/IP Properties. Listen all oznacza, że port nasłuchuje na wszystkich kartach sieciowych. IP Addresses - dokładniejsza konfiguracja. Jeśli SQL Server nie nasłuchiwał na wszystkich kartach sieciowych, to należałoby każdy interfejs sieciowy skonfigurować oddzielnie. Jeśli nasłuchuje na wszystkich to patrz na ostatnią część, czyli IPAll. TCP Port - oznacza na którym porcie nasłuchuje SQL. Można tą wartość zmienić. Jeśli jednak chcesz żeby SQL za każdym razem nasłuchiwał na innym porcie w TCP Dynamic Ports wpisz zero (0) - uwaga, ta konfiguracja powoduje, że trudniej jest konfigurować firewalle.

**Opcje konfiguracji klientów** → zmiany wyjątkowo w SQL Native Client 32bit. SQL Server management Studio i Visual Studio są programami 32-bitowymi. Aby klient mógł połączyć się z serwerem musi mieć te same protokoły albo przynajmniej jeden wspólny protokół co jest na serwerze. Client Protocols → sensownym rozwiązaniem jest mieć włączone wszystkie protokoły klienckie. Kolumna order pokazuje w jakiej kolejności protokoły będą używane. Oznacza to, że klient łączący się z serwerem najpierw zrobi to (na przykładzie podanym) najpierw przez Shared memory (pamięć wspólna na komputerze), potem przez TCP/IP, a na końcu przez Named Pipes. Zakładka Aliases: pozwala zdefiniować alias pod którym będzie dany serwer.

**Tworzenie nowego Aliasu** → Native Client 32bit, Aliases, New Alias, nazwij go jakoś, w server wpisz nazwę twojego servera (na przykładzie DESKTOP-FLCPP4I), Ok. Teraz podłącz się do serwera za pomocą nazwy twojego aliasu. Tak naprawdę podłączyłeś się do DESKTOP-FLCPP4I. Tak stworzony alias musi być utworzony na kliencie, który łączy się do SQL Servera.

**7. Bazy systemowe**

**System databases** → bazy systemowe znajdujące się na każdej instancji SQL Server. Ścieżka: databases, system databases.

**Master** → najważniejsza baza danych. Przechowuje wszelkie definicje, które pozwalają pracować SQL Serverowi, np. informacje jakie inne bazy danych są na tym serwerze dostępne.

**Views** → folder przechowujący informacje systemowe. Dostęp: Databases, system databases, Maser, Views.

**Odwoływanie się do konkretnych baz danych** → poniżej znajdują się różne dostępy do informacji baz danych: master, model, msdb, tempdb (przy czym AdventureWorsk jest również bazą danych. Zwróć uwagę na ikony w SQL, są takie same). Żeby podejrzeć odpowiednie tabele (za pomocą SELECT) albo pisać procedury (za pomocą exec) to pamiętaj, że odwołujesz się do określonej bazy. Oznacza to, że musisz ją wskazać albo za pomocą USE albo dodając na początku manipulowanej tabeli “nazwa\_tabeli.”. Np. SELECT \* FROM msdb.dbo.restorehistory).

**Korzystanie z widoków systemowych** →

1. sys.databases → widok zawierający informacje jakie bazy danych zostały zainstalowane na tym SQL Serverze. Ponadto pokazuje szczegółowe informacje o bazach danych.

Np.

SELECT \* FROM sys.databases

1. sys.server\_principals → pozwala uzyskać informacje o istniejących na serwerze loginach.

Np.

SELECT \* FROM sys.server\_principals

1. sys.database\_principals → odpowiada do użytkowników bazy danych zmapowanych do loginów.

Np.

USE AdventureWorks

SELECT \* FROM sys.database\_principals

1. sys.dm\_exec\_connections → przedstawia połączenia do SQL Servera. Połączanie lub kończenie sesji pracy z SQL Serverem tutaj powinno być odzwierciedlone. Każda sesja ma nadany numer. Każda sesja powstaje wskutek połączenia się określonym protokołem sieciowym, np. Shared Memory. Numer twojej sesji widać na końcu tytułu otwartego skryptu. Kolumny last read i last write pozwalają określić czy sesja jest aktywna czy nie. Kolumna most\_recent\_sql\_handle pozwala określić jakie ostatnio polecenie SQL było używane.

Np.

SELECT \* FROM sys.dm\_exec\_connections

**Funkcja sys.dm\_exec\_sql\_text** → wklej informację uzyskaną w most\_recent\_sql\_handle (patrz wyżej) do funkcji sys.dm\_exec\_sql\_text (argument), a dowiesz się jakie ostatnio polecenie było wykonywane w ramach tej sesji. W ten sposób jesteś w stanie wyśledzić aktywność użytkowników. W poniższym przykładzie jest aktywność z sesji kiedy był ten przykład tworzony i nie użyjesz go tak po prostu. Musisz wydobyć nową aktywność.

Np.

SELECT \* FROM sys.dm\_exec\_sql\_text(0x02000000FEC7CB19E3AB91BD34F4A2654EEC3AE7DADD82C50000000000000000000000000000000000000000)

**Procedura dbo.xp\_fixeddrives** → informuje ile jest miejsca na określonych dyskach na serwerze. Piszesz ją samą, bez żadnego SELECT. Dodaj “exec” i “master” na początku żeby procedura zadziałała.

Np.

exec master.dbo.xp\_fixeddrives

**Funkcja dbo.sp\_who** → dowiedz się kto pracuje na twoim serwerze.

Np.

exec dbo.sp\_who

**Właściwości bazy danych master** → odnajdziesz tam lokalizację pliku i lokalizację pliku log: czyli parametry startowe d i l (patrz wyżej). Ścieżka: system databases, master, files.

**Baza danych model** → modelowa baza danych. Jeśli wydajesz polecenie Create Database to tworzysz kopię tej bazy danych. Jeśli chcesz żeby nowe bazy miały określone opcje lub obiekty, to utwórz je w bazie model.

**Baza msdb →** baza ta wspiera pracę SQL Server Agenta. Znajdziesz tam informacje o jobach i innych aktywnościach (backupów, odzyskiwaniu baz danych). Jeśli skonfigurowałeś bazę do wysyłania maili to w bazie danych znajdziesz kopię tych maili. Warto wykonywać tu zadania, które czyszczą bazy i usuwają niepotrzebne historyczne rekordy.

**Funkcja dbo.restorehistory** → mówi kiedy dochodziło do odtworzenia bazy danych. Pamiętaj, że używasz tej funkcji na bazie msdb.

Np.

SELECT \* FROM msdb.dbo.restorehistory

**Funkcja dbo.sysjobs** → jakie zadania są aktywne w SQL Server Agencie.

Np.

USE msdb

SELECT \* FROM dbo.sysjobs

**Praca na określonej bazie danych** → w rameczce w lewym górnym rogu widać na jakiej bazie aktualnie pracujesz.

**Baza tempdb** → baza danych potrzebna do przechowywania obiektów tymczasowych podczas pracy w SQL Serwerze. Np. tabele użytkowników (tymczasowe) - ich nazwa zaczyna się od znaku #. Tak powstaje tabela tymczasowa.

Np.

CREATE TABLE #temp (id int)

W temp.db powstają też obiekty, które powstają podczas pisania zapytań, kiedy SQL Server stwierdza, że chciałby tymczasowo trzymać dane na dysku. Można tą bazę wykorzystać też do budowania indeksów, itd.

Właściwości: powinno być tyle plików temp.db ile jest dostępnych procesorów, do około 8. Ta baza danych powinna mieć cały dysk i to najszybszy. Sprzyja to szybkiemu tworzeniu zapytań. Plik loga za to wystarczy tylko jeden. Ta baza danych powstaje od nowa przy każdym otwarciu SQL Servera. Nie ma sensu tworzyć backupu tej bazy.

**8. SQL Database w Microsoft Azure**

**Microsoft Azure** → chmura Microsoft. Można utworzyć w chmurze czyli serwerze swoje dane, też za pomocą SQL Server. Jest to usługa płatna.

**Tworzenie obiektów** → jest łatwe, ale do żadnego widoku nie ma się co przyzwyczaić. Często się zmienia. Tworzysz nową bazę danch SQL Server. Użyj serwera baz danych. Tworzysz obiekt SQL Database. Dajesz nazwę. Określasz serwer na jakim baza ma się znajdować. Create new server. Nazwij go i nazwę administratora, któr będzie miał dostęp i jego hasło. Określ gdzie ma być serwer. Może to być wszędzie na świecie. Tworzysz pustą (blank) bazę danych. Określasz ile płacisz - chodzi o wielkość i ilość transakcji. Jako test będziesz wybierać basic, najmniejszą opcję. Określasz też collation (sortowanie) bazy danych. Create - chwilę to potrwa.

**Konfiguracja i uruchomienie bazy danych** → aby podłączyć się do nowej bazy użyj jej Server name. Jeśli podłączasz się z aplikacji to użyj gotowych Connection string. Wklejasz go do aplikacji. Skopiuj adres serwera, uruchom Management Studio, wklej adres. Uwierzytelnienie Azure wymaga uwierzytelnienie SQLowe. Adres musi być też dodany do białej listy firewalla. Co dalej, patrz Dodawanie wyjątku do firewalla.

**Dodawanie wyjątku do firewalla** → w opcjach (Azure) jest opcja pozwalająca zarządzać firewallem. Add client IP. Mój adres IP został rozpoznany i dodany do białej listy. Save. Teraz podłącz się do SQL. Connect.

**Zarządzanie serwerem Azure w SQL** → jest o wiele uboższe. Mogę tylko dodawać kolejne bazy. Nie ma interfejsu graficznego. Nie konfigurujesz plików dodawanych do bazy danych - to robi Microsoft. Nie ma też properties bo tym też zarządza Microsoft. Masz do dyspozycji bazę Master i twoją bazę danych. Nie możesz używać polecenie USE - zamiast tego przełączaj się między bazami za pomocą rozwijanego słupka. Możesz dodawać dane i pracujesz jakbyś pracował z lokalnym serwerem i lokalną bazą danych.

**Korzystanie z Azure** → korzystaj gdy twoja aplikacja będzie używana krótko, sezonowo. W zamian Microsoft stara się o wysoką dostępność i wydajność.

**9. Budowanie zapytań**

**Przykładowa modyfikacja zapytania** → wyświetl zawartość tabeli, np. Person.Person przez Select Top 1000 Rows. Górna część to wygenerowane zapytanie, a na dole jego wynik. Możesz modyfikować ilość wyświetlanych rekordów. Możesz zmieniać wygenerowane zapytanie, np. połączyć dwie kolumny w jedną, np. First Name i Last Name w jeden napis. Zrób to w następujący sposób: do ,[FirstName] dodaj +’ ‘+ [LastName] AS FullName. Pojawia się konstrukcja FullName. Możesz ograniczyć listę rekordów do tych, które spełniają określony warunek. Napisz na końcu np. WHERE Title =’Mr.’. Usuń zapis TOP 100 żeby zobaczyć wszystkie rekordy z tabeli. Pamiętaj żeby nie rezygnować z zapisu TOP gdy pracujesz np. z danymi produkcyjnymi, gdyż tych rekordów może być tak dużo żeby zawiesić program lub serwer.

Np.

SELECT TOP (100) [BusinessEntityID]

   ,[PersonType]

   ,[NameStyle]

   ,[Title]

   ,[FirstName] + ' ' + [LastName] AS FullName

   ,[MiddleName]

   ,[LastName]

   ,[Suffix]

   ,[EmailPromotion]

   ,[AdditionalContactInfo]

   ,[Demographics]

   ,[rowguid]

   ,[ModifiedDate]

  FROM [AdventureWorks].[Person].[Person]

  WHERE Title = 'Mr.'

**Omówienie wzoru SELECT** → USE Baza danych SELECT TOP (ile top) [Kolumna\_1], [Kolumna\_2], [Kolumna\_do\_łączenia\_1] + ‘jakiś symbol łączenia’ + [Kolumna\_do\_łączenia\_2] AS [Nowa\_kolumna] FROM Tabela WHERE warunek

Np.

USE AdventureWorks

SELECT TOP (100)

[DepartmentID],

[Name],

[GroupName],

[Name] + ' ' + [GroupName] AS [FullName],

[ModifiedDate]

FROM HumanResources.Department

WHERE DepartmentID > 10

**INNER JOIN →** polecenie łączące tabele. Przy czym pamiętaj, że odwołując się do nazw kolumn, nie wystarczy podać ich nazw. Przed nimi powinna być najpierw nazwa schematu i tabeli, po kropkach - jest tak dlatego, że w tabelach nazwy kolumn mogą się powtarzać i nie wiadomo by było o jaką kolumnę chodzi.

INNER JOIN wybiera rekordy, które mają pasujące wartości w obu tabelach. Wzór: SELECT Kolumny, które cię interesują FROM tabela\_1 INNER JOIN tabela\_2 ON klucz\_podstawowy\_tabeli\_1 = klucz\_podstawowy\_tabeli\_2

Np.

USE AdventureWorks

SELECT Person.BusinessEntityAddress.rowguid, Person.BusinessEntityAddress.ModifiedDate, Person.AddressType.Name

FROM Person.BusinessEntityAddress INNER JOIN Person.AddressType

ON Person.BusinessEntityAddress.AddressTypeID = Person.AddressType.AddressTypeID

**AS** → Polecenie AS służy do zmiany nazwy kolumny lub tabeli za pomocą aliasu. Alias istnieje tylko na czas trwania zapytania.

Np.

USE AdventureWorks

SELECT PersonType AS Person, NameStyle AS Name

FROM Person.Person

**ON** → Klauzula używana do zdefiniowania warunku złączenia (podczas gdy WHERE powinno być użyte do filtrowania danych - taka jest różnica między nimi). Użyłem słowa „powinien”, ponieważ nie jest to trudna zasada. Podział tych celów z ich odpowiednimi klauzulami sprawia, że zapytanie jest najbardziej czytelne, zapobiega również pobieraniu nieprawidłowych danych podczas korzystania z typów JOIN innych niż INNER JOIN.

Np.

USE AdventureWorks

SELECT Person.BusinessEntityAddress.rowguid, Person.BusinessEntityAddress.ModifiedDate, Person.AddressType.Name

FROM Person.BusinessEntityAddress INNER JOIN Person.AddressType

ON Person.BusinessEntityAddress.AddressTypeID = Person.AddressType.AddressTypeID

**ORDER BY … ASC** → Polecenie ORDER BY służy do sortowania zestawu wyników w kolejności rosnącej lub malejącej. Polecenie ORDER BY domyślnie sortuje zestaw wyników w kolejności rosnącej. Aby posortować rekordy w kolejności malejącej, użyj słowa kluczowego DESC. Możesz też oczywiście wpisać ASC jeśli to zapewni ci większą czytelność.

Np.

SELECT \* FROM Person.Person

ORDER BY FirstName ASC

– Posortuje od A do Z

**ORDER BY … DESC** → sortuje w kolejności malejącej.

Np.

Np.

SELECT \* FROM Person.Person

ORDER BY FirstName DESC

– Posortuje od Z do A

**Budowanie zapytania za pomocą Query Designer** → jest to narzędzie, które pozwala w uproszczony sposób tworzyć zapytania. Przydatne dopóki nie umiesz jeszcze pisać kodu samemu. Ścieżka: tworzysz nowy skrypt, naciskasz na jego przestrzeń prawym i “Design Query in editor”. Dodajesz tabele, które mają powiązania za pomocą klucza.

Na przykładzie Product Category i Product Subcategory. Dodaj je. Chcę osiągnąć, żeby była jedna tabela z kolumnami: ProductCategoryID, Name, ProductSubcategoryID, Subcategory. W treści zapytania na dole już pojawia się polecenie INNER JOIN, które służy do łączenia tabel. Zaznacz kolumny, które chcesz żeby zapytanie wyświetlało - zwróć uwagę, że dodajesz ProductCategoryID, Name (z ProductCategory), ProductSubcategoryID, Name (z ProductSubcategory, którego nazwę zaraz zmienisz). Dodają się kolumny i w kodzie też.

Wzór: SELECT kolumny, gdzieś tam AS zmieniające nazwę jednej kolumny, FROM nazwa jednej, ważniejszej tabeli i INNER JOIN nazwa mniej ważnej tabeli ON klucz podstawowy tabeli wspólny 1 = klucz podstawowy wspólny tabeli 2.

Ok i Execute. Możesz powstały kod zmieniać w edytorze. Np. zmień Name AS Expr1 na coś innego (np. Subcategory). Jeśli dalej chcesz dokonywać zmian w Query Editor to zaznacz cały kod i uruchom Edytor. Możesz dodawać filtry rekordów (Filter), np. dla Name daj = N’Bikes’ lub po prostu ‘Bikes’ (co doda WHERE (nazwa kolumny = N’Bikes’), program sam doda N. Dodaj sortowanie i określ jak ma się odbywać (Sort type) przez Subcategory (co doda ORDER BY Subcategory).

Np.

SELECT    Production.ProductCategory.ProductCategoryID, Production.ProductCategory.Name, Production.ProductSubcategory.ProductSubcategoryID, Production.ProductSubcategory.Name AS Subcategory

FROM        Production.ProductCategory INNER JOIN

                      Production.ProductSubcategory ON Production.ProductCategory.ProductCategoryID = Production.ProductSubcategory.ProductCategoryID

WHERE    (Production.ProductCategory.Name = N'Bikes')

ORDER BY Subcategory

**10. Modyfikacja danych w SQL Management Studio**

**Dodawanie / edytowanie / modyfikacja danych w SQL** → wybierz tabelę i edit TOP 200. Możesz jak w Excel modytfikować dane. Po prostu je podmieniasz. Inaczej jest w przypadku dodania rekordu - robisz to od dołu. Dodaj wszytko poza Modified Date i ID. Te dane wypełni sam SQL. Musisz jednak nacisnąć Execute SQL po zakończeniu dodawania rekordu (Crtl+R). Usuwanie za pomocą prawy na rekord i Delete.

**Execute SQL →** komenda potwierdzająca zmianę. Crtl+R. Wykorzystaj np. do dodaniu nowego rekordu.

**Modyfikacja tabeli z większą ilością rekordów** → trzy przyciski cię tu interesują.

Show Diagram Pane (Crtl+1), który pokazuje które kolumny program ma wyświetlać (pamiętaj żeby potwierdzić zmiany Execute SQL);

Show Criteria Pane (Crtl+2), który pokazuje wg. jakich kryteriów program pokazuje dane;

Show SQL Pane (Crtl+3), który pokazuje kod. W tym ostatnim możesz zmienić TOP 200 na cokolwiek innego.

Np.

SELECT TOP (200) BusinessEntityID, Name

FROM Purchasing.Vendor

WHERE (Name = N'Wood Fitness')

ORDER BY BusinessEntityID DESC

Pamiętaj jednak, że powodem dla którego domyślne jest TOP 200 jest fakt, że czasem tabela jest tak duża, że wyświetlenie wszystkich rekordów może być zbyt obciążające dla sprzętu. Dlatego rekomenduje się modyfikować dane na podstawie określonych kryteriów, co można ustalić w powyższych trzech opcjach. Pamiętaj na końcu o Execute SQL.

**Funkcja LIKE** → Operator LIKE jest używany w klauzuli WHERE do wyszukiwania określonego wzorca w kolumnie. Istnieją dwa symbole wieloznaczne często używane w połączeniu z operatorem LIKE:

Znak procentu (%) reprezentuje zero, jeden lub wiele znaków

Znak podkreślenia (\_) reprezentuje jeden, pojedynczy znak

Np.

USE AdventureWorks

SELECT \* FROM Sales.Currency

WHERE (Name LIKE 'A%')

– Zwraca wszystko co zaczyna się na A

WHERE (Name LIKE 'Af%')

– Zwraca wszystko co zaczyna się na Af

WHERE (Name LIKE 'Afghan\_')

– Zwraca wszystkie rekordy z Name = Afghani, dlatego, że \_ zastępuje tylko jedną literę

WHERE (Name LIKE 'Afgha\_\_')

– Zwraca wszystkie rekordy z Name = Afghani, dlatego, że \_\_ zastąpią dwie litery

**11. Modyfikacja danych**

**Tworzenie skryptów z pomocą SQL** → zanim nauczysz się płynnie pisać kod, korzystaj z podstawowych funkcji SQL.

**GetDate()** → funkcja podająca teraźniejszą datę i godzinę.

**INSERT INTO** → dodaje dane do tabeli. Kod wybiera bazę danych. Wybiera tabelę w której dodaje

        (<Name, [dbo].[Name],>

        ,<StartTime, time(7),>

        ,<EndTime, time(7),>

        ,<ModifiedDate, datetime,>)

GO

Prawym na tabelę na której chcesz pracować i Script Table As. my rekord. Wybiera kolumny, które będą wypełniane. Zwróć uwagę, że nie ma tam tabeli ID, która będzie wypełniona automatycznie przez SQL Server. Teraz skorzystaj z Crtl+Shift+M w celu dodania rekordu. W przypadku tekstu, dat i godziny pamiętaj o apostrofach. Datę modyfikacji określ na teraźniejszą za pomocą funkcji GetDate()

Np.

USE [AdventureWorks]

GO

INSERT INTO [HumanResources].[Shift]

        ([Name]

        ,[StartTime]

        ,[EndTime]

        ,[ModifiedDate])

  VALUES

**VALUES** → komenda, która precyzuje w INSERT INTO jakie wartości mają być dodane. Pamiętaj, że nie musisz dodawać wartości do każdej kolumny, niektóre mogą mieć NULL. Jeśli jakieś nie mogą mieć to program cię o tym poinformuje.

**Wzór kodu INSERT INTO** → INSERT INTO [schemat].[tabela] ([kolumna\_1], [kolumna\_2]) VALUE (wartość kolumny 1, wartość kolumny 2) GO

Np.

USE AdventureWorks

SELECT \* FROM Production.Product

WHERE Name = 'Chainsaw'

GO

INSERT INTO [Production].[Product]

([Name]

,[Color]

,[ProductNumber]

,[SafetyStockLevel]

,[ReorderPoint]

,[StandardCost]

,[ListPrice]

,[DaysToManufacture]

,[SellStartDate]

,[ModifiedDate])

VALUES

('Chainsaw'

,'Silver'

,'CA-2222'

,1000

,600

,0

,0

,1

,GetDate()

,GetDate())

GO

**SELECT** → wyświetli dane jak w przykładzie poniżej. Kod najpierw wybierze bazę danych, na których pracuje, potem określi jakie kolumny pokaże i z jakiej tabeli. Wzór: SELECT [Kolumna\_1], [Kolumna\_2] FROM [Schemat].[Tabela]

Np.

USE [AdventureWorks]

GO

SELECT [ShiftID]

   ,[Name]

   ,[StartTime]

   ,[EndTime]

   ,[ModifiedDate]

  FROM [HumanResources].[Shift]

GO

**Nawiasy kwadratowe w SQL** → SQL ukrywa niektóre dane, np. nazwy kolumn z nawiasach kwadratowych w razie gdyby przypadkiem były one zbieżne z elementami kody Transact SQL. Powstały kod możesz dowolnie modyfikować.

**UPDATE** → modyfikuje dane w tabeli. Najpierw wybiera bazę potem tabelę do modyfikacji. Następnie za pomocą funkcji SET wybierasz które dane, z których kolumn chcesz modyfikować. Protip: jeśli chcesz zmodyfikować tylko niektóre dane z niektórych kolumn to usuń te linie kodu. Alternatywnie możesz np. zrobić [StartTime] = [StartTime], ale to wprowadzi niepotrzebnie zamieszanie. Na końcu WHERE podaje które rekord, który chcesz modyfikować. Znowu za pomocą Crtl+Shift+M. Pamiętaj, że w Search Conditions piszesz Nazwa kolumny = ‘Stara nazwa’ (np. Name = 'On Duty')

Np.

USE [AdventureWorks]

GO

UPDATE [HumanResources].[Shift]

   SET [Name] = <Name, [dbo].[Name],>

      ,[StartTime] = <StartTime, time(7),>

      ,[EndTime] = <EndTime, time(7),>

      ,[ModifiedDate] = <ModifiedDate, datetime,>

 WHERE <Search Conditions,,>

GO

**UPDATE →** komenda używana do aktualizacji istniejących rekordów w tabeli.

**UPDATE TO kod** → UPDATE [Schemat].[Tabela] SET [Kolumna] = Nowa wartość, [Kolumna\_2] = Nowa wartość\_2 WHERE określasz jaki rekord zmieniasz, najlepiej kluczem podstawowym. Pamiętaj, że we wcześniejszych warunkach po SET nie musisz nic podawać o jaki rekord chodzi. To robisz po WHERE.

Np.

USE [AdventureWorks]

GO

UPDATE [HumanResources].[Department]

   SET [Name] = 'HR\_2'

   ,[GroupName] = 'Soft\_2'

   ,[ModifiedDate] = GetDate()

 WHERE DepartmentID = 21

GO

**DELETE** → usuwanie rekordu. Nazwa bazy danych, i tabela, a następnie który rekord usunąć. Wykorzystaj ID. Znowu należy wpisać Nazwa kolumny = ‘Wartość’ (np. WHERE ShiftID = '27'). Dobrą praktyką jest w celu usuwania rekordów najpierw stwórz SELECT razem z WHERE, a potem podmień SELECT \* FROM na DELETE FROM.

Np.

USE [AdventureWorks]

GO

DELETE FROM [HumanResources].[Department]

   WHERE DepartmentID = 21

GO

**Funkcja IN** → funkcja w połączeniu z WHERE pozwala na np. w SELECT na pokazanie więcej niż jednego, konkretnego rekordu.

Np.

USE [AdventureWorks]

GO

SELECT \* FROM HumanResources.Shift

WHERE ShiftID IN ('25', '26')

**12. Projektowanie tabeli**

**Tworzenie tabeli za pomocą interfejsu graficznego** → utwórz schemat. Schematy grupują tabele i pozwalają na łatwiejsze zarządzanie uprawnieniami do modyfikowania. Jeśli użytkownicy mają mieć uprawnienia do modyfikowania tabel z określonego schematu, to nie musisz nadawać uprawnień indywidualnie. Schematy znajdziesz w Adventure Works, Security, schemas. Utwórz nowy schemat, New Schema. Nadaj mu nazwę i Script (żeby podejrzeć skrypt).

Np.

USE [AdventureWorks]

GO

CREATE SCHEMA [Dojazdy]

GO

Następnie prawym na węzeł Tables i Table. Na przykładzie stwórz tabelę o garażach. Dodaj nazwę “GarageID” i nadaj typ int, jak integer, czyli liczby całkowite - to pozwala na wstawianie dużej liczby rekordów. Dla każdej tabeli możesz ustalić wartość NULL - w tym przypadku chcę, żeby każdy rekord był identyfikatorem, więc nie zezwalam na NULL, użytkownik zawsze musi coś wstawić. Chcę też, żeby wartość ta sama się wypełniła. W Column Properties znajdź Identity Specification i zmień Is Identity na Yes. Teraz wartość będzie zaczynała się od 1 (Identity Seed) i kolejna nowa wartość będzie miała wartość plus 1 (Identity Increment). Każda tabela może mieć tylko jedno pole, które zostało oznaczone jako Identity.

Stwórz kolejną kolumnę, np. o nazwie “Description”, która oznaczy garaż, który wprowadzam do tabeli. Pozwól na opis do 100 znaków. Typ = nchar(100) i nie pozwól na NULL.

Dodaj też oczywiście ModifiedDate. Zadbam na razie tylko o datę i godzinę dodania rekordu i żeby była wprowadzana automatycznie, jeśli użytkownik sam tego nie zrobił. Użyj datetime2(7), który wybierze datę z dość dużą dokładnością. Nie zezwolę na NULL. Dodatkowo muszę wprowadzić wartość domylną jeśli użytkownik sam nie wpisze nic: General, Default Value or Binding SYSDATETIME(). Program sam doda jeszcze jeden nawias.

**Funkcja SYSDATETIME()** → zwraca bieżący czas w formie datetime2, czyli dokładną datę i czas. Wpisując do Default Value or Binding, piszesz (SYSDATETIME())

**Tworzenie tabeli za pomocą interfejsu graficznego, cd.** →

Dodaj nazwę tabeli: properties (otwierasz za pomocą ikony klucza), Name. Z nazwą pamiętaj zastanowić się czy nazwa w liczbie pojedynczej czy mnogiej. Zależy od stylu bazy danych na której pracujesz - tutaj pojedyncza.

Utworzenie klucza podstawowego: prawym na kolumnę, która ma zawierać te klucze i set primary key - wtedy dopiero jest kluczem podstawowym.

Ustalenie schematu, w którym ma być tabela: schemat musi już istnieć

ustal w jakim schemacie ma być tabela. properties: schema i wpisz nazwę schematu.

Zapis tabeli: za pomocą Crtl+S lub lepiej podejrzeć skrypt. Generate script - pamiętaj jednak, że schemat, który wybrałeś musi już istnieć i być zaktualizowany/odświeżony jeśli nowy..

Np.

/\* To prevent any potential data loss issues, you should review this script in detail before running it outside the context of the database designer.\*/

BEGIN TRANSACTION

SET QUOTED\_IDENTIFIER ON

SET ARITHABORT ON

SET NUMERIC\_ROUNDABORT OFF

SET CONCAT\_NULL\_YIELDS\_NULL ON

SET ANSI\_NULLS ON

SET ANSI\_PADDING ON

SET ANSI\_WARNINGS ON

COMMIT

BEGIN TRANSACTION

GO

CREATE TABLE guest.Garage

(

GarageID int NOT NULL IDENTITY (1, 1),

Description nchar(100) NOT NULL,

ModifiedDate datetime2(7) NOT NULL

)  ON [PRIMARY]

GO

ALTER TABLE guest.Garage ADD CONSTRAINT

DF\_Garage\_ModifiedDate DEFAULT (SYSDATETIME()) FOR ModifiedDate

GO

ALTER TABLE guest.Garage ADD CONSTRAINT

PK\_Garage PRIMARY KEY CLUSTERED

(

GarageID

) WITH( STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

GO

ALTER TABLE guest.Garage SET (LOCK\_ESCALATION = TABLE)

GO

COMMIT

Okno dialogowe pokazuje jaki będzie skrypt.

Komentarz: Create table (tworzy tabelę jaką). W następnym nawiasie są nazwy kolumn, które stworzyłem i ich właściwości.

Alter table modyfikuje tabelę np. dodając wartość domyślną dla ModifiedDate - każdy obiekt w SQL Server musi mieć swoją nazwę. Tutaj będzie ona być wyznaczona automatycznie (tutaj DF\_Garage).

Primary key oznacza utworzenie klucza podstawowego. On też ma ustaloną domyślną nazwę: tutaj PK\_Garage.

Zapisz i zrób refresh w Tables. Są tam kolumny, klucze, constraints - który zadba, żeby pojawiała się data wstawienia rekordu. Dodaj rekord za pomocą Insert i nie podawaj modified date.

**13. Budowanie relacji między tabelami**

**Definiowanie relacji w bazie danych** → potrzebne co najmniej dwie tabele

Tworzysz nową tabelę i ona musi mieć takie samo jedno ID jak ta z którą będziesz tworzyć relacje. Jednak tej nowej tabeli nadajesz inny klucz podstawowy. Przy tworzeniu bazy danych użyj opcji Relationships (ikona dwóch połączonych tabel). Po kliknięciu Add pojawia się opis pierwszej relacji. Rozwiń Tables and Columns Specification - tu określisz, która kolumna tej tabeli odnosi się do której kolumny innej tabeli. Korzystaj z ikony wielokropka przy pasku Tables and Columns Specification. Wybierz tabelę i jej klucz podstawowy. Nie możesz zmienić drugiej tabeli w relacji, ale wskazujesz kolumnę wskazującą do tej pierwszej. Klikasz OK i sprawdzasz w tables and columns specification czy relacja jest poprawna. Pojawia się też nazwa dla klucza podstawowego zaczynająca się od FK jak Foreign Key i dwie nazwy tabel. Teraz zamknij okno i wygeneruj skrypt dla całej tabeli - będzie tam też skrypt tworzenia nowej relacji. Skopiuj skrypt do nowego okna i uruchom go ręcznie. Sprawdzanie relacji za pomocą new diagram. Możesz z pola bocznego przeciągnąć interesujące cię tabele.

Np.

/\* To prevent any potential data loss issues, you should review this script in detail before running it outside the context of the database designer.\*/

BEGIN TRANSACTION

SET QUOTED\_IDENTIFIER ON

SET ARITHABORT ON

SET NUMERIC\_ROUNDABORT OFF

SET CONCAT\_NULL\_YIELDS\_NULL ON

SET ANSI\_NULLS ON

SET ANSI\_PADDING ON

SET ANSI\_WARNINGS ON

COMMIT

BEGIN TRANSACTION

GO

ALTER TABLE guest.Garage SET (LOCK\_ESCALATION = TABLE)

GO

COMMIT

BEGIN TRANSACTION

GO

ALTER TABLE Office.ParkingPlace ADD CONSTRAINT

FK\_ParkingPlace\_Garage FOREIGN KEY

(

GarageId

) REFERENCES guest.Garage

(

GarageID

) ON UPDATE  NO ACTION

ON DELETE  NO ACTION

GO

ALTER TABLE Office.ParkingPlace SET (LOCK\_ESCALATION = TABLE)

GO

COMMIT

**Komentarz** → ostatnie AlterTable to dodanie klucza obcego. Nazwa klucza i Foreign Key za Alter table i add constraint. References czyli do czego się odnosi.

**Bit** → typ danych bit, może przechowywać 0, 1 lub NULL

**Default Value or Binding** → to wartość domyślna w rekordzie.

**nvarchar(50)** → typ zmiennych, które przechowują do 50 znaków. Możesz tu wprowadzić inną liczbę, np. 100, 123, 45, 19…

**14. Skryptowanie bazy danych i jej obiektów**

**Skryptowanie**→ przydaje się, gdy chcesz wykonać operację i mieć nad nią jak najlepszą kontrolę oraz mieć ślad co się zmieniło. Skryptujesz za pomocą np. Script table as, Create to - dzięki temu poznasz skrypt wykorzystany do utworzenia konkretnej tabeli.

**Usuwanie tabeli** → za pomocą Drop, które jest jedną z opcji skryptowania. Możesz też prawym na tabelę i Delete. Dobra praktyka - do usuwania twórz skrypt raczej niż delete. Dzięki temu zostaje ślad, że to zrobiłeś i co usunąłeś. Script table as, Drop to.

**Skryptowanie bazy danych i wszystkich obiektów w niej się znajdujących** → AdventureWorks, Tasks, Generate Scripts. Określ co chcesz zeskryptować z bazy lub automatycznie wybierz wszystko. Kolejny krok: decydujesz jak i gdzie utworzyć skrypt. Dla każdego obiektu w osobnym pliku czy razem?

Advanced: zmiania sposób w jaki skrypt jest generowany. Np. Check for object existence: w skrypcie będą bloki IF sprawdzające czy dany obiekt już w bazie jest.

Script DROP and CREATE: określa czy wybrane obiekty są usuwane (DROP) czy dodawane (CREATE).

Script for Server Version: wybierasz wersję SQL Server dla którego skrypt ma być utworzony. pamiętaj jednak, że jeśli skorzystasz np. z nowego typu obiektu, którego w starszych wersjach nie było, to on się nie utworzy.

Skryptować można też uprawnienia - kto może korzystać.

Types of data script: ważne, że możesz oskryptować nie tylko tabelę, ale też dane, które ją wypełniają: Types of data to script: Schema and data - wyskryptuje się seria poleceń Insert, które wypełniają tabelę. Skrypt tabeli wypełni się wartościami, które można dodać.

Table/view options: kategoria opcji “Table/View Options” pokazuje różne opcje tabeli typu definiowanie klucza, które mogą być oskryptowane.

W następnym kroku kreatora weryfikujesz czy wszystko zostało wybrane poprawnie.

Wygeneruj skrypt. Otwórz go za pomocą Open File (Crtl+O).

**Otwieranie pliku** → open file, Otwieranie utworzonego/istniejącego skryptu, Crtl+O.

**Przenoszenie bazy danych z nowszej do starszej wersji** → przenoszenie bazy danych z nowszej do starszej wersji należy to właśnie zrobić tak, za pomocą skryptów. Back up z wyższej wersji SQL Server nie będzie działać w nowszej wersji. Dlatego skryptujesz i przenosisz treść skryptu.

**SQL Server - zapytania, querying**

**15. Wprowadzenie do SELECT**

**Komentowanie w kodzie** → komentowanie ma dwie postacie:

1. Linia zaczęta – (dwa myślniki);
2. Blok komentarza:

/\*

treść

\*/

**Schemat** → grupuje tabele. Schematy służą do zarządzania uprawnieniami.

**Podgląd tabel →** skorzystaj z eksploratora tabel, żeby podejrzeć kolumny tabeli i typy zmiennych, które znajdują się w kolumnach.

**SELECT \* FROM** → czyli select gwiazdka nadaje się do testowania, nauki czy pierwszego podejrzenia bazy, ale do oddawania programu lepiej wymieniać poszczególne, potrzebne kolumny.

**Przecinki w kodzie** → powinny pojawiać się na początku linijki kodu, np. przy wymienianiu kolumn do wygenerowania. Łatwiej je na chwilę lub ogólnie usunąć przez zakomentowanie, a poza tym czytelniejsze jest gdzie zaczynają się i kończą wymieniania kolumn.

**Słowa w kodzie zaznaczone na niebiesko** → słowa kluczowe, które w inny miejscu miałyby specjalne znaczenie zarezerwowane dla SQL Server. Możesz umieszczać je w nawiasach kwadratowych co oznacza, że pozbawiasz je tego specjalnego znaczenia. Przykład takich słów kluczowych: Name, Weight.

**Sugestia tabel w schemacie** → zacznij pisać schemat kropka. Jeśli pod SELECT zaczniesz pisać to program będzie podpowiadać pisząc pierwsze litery.

**AS** → nadawanie aliasu tabeli, co przyspiesza pisanie kodu SELECT dla wybranych kolumn. Zamiast pisać całą nazwę, teraz możesz korzystać z aliasu - robisz to bo dzięki temu program podpowie całą listę kolumn i tylko je wybierasz. Zaczynasz od końca czyli od FROM żeby to działało.

Np.

USE AdventureWorks

SELECT

    p.FirstName

    ,p.Demographics

    ,p.NameStyle

FROM Person.Person AS p

GO

**AND i OR. Warunki filtrujące określone rekordy** → jeśli chcę wyświetlać określone rekordy (wiersze) dla wybranych kolumn. Dodawanie kolejnych warunków, dobudowanie ich odbywa się za pomocą AND. Możesz dla liczb korzystać z większe/mniejsze/równe/itp. Za pomocą OR oddzielasz poprzedni warunek złożony z różnych AND i piszesz kolejny warunek.

USE AdventureWorks

SELECT

\*

FROM Person.Person

WHERE

MiddleName = 'A'

AND Title = 'Ms.'

OR LastName = 'Duffy'

GO

– Wyświetli wyniki dla dwóch warunków. Program odda wszystko co pasuje w warunku pierwszym i drugim.

Odwoływanie się do bazy danych z której nie korzystasz → jeśli jest taka potrzeba, to przykładowo w SELECT wygląda to tak, że podajesz za FROM najpierw baza danych.schemat.tabela.

Np.

USE master

GO

SELECT \* FROM AdventureWorks.Sales.SalesPerson

**16. Klauzula WHERE I**

**GO** → Uruchamiając cały skrypt, program uruchamia każdy jego fragment po kolei - fragment rozumie się tu przez część do kolejnego GO i potocznie ta część nazywa się batch (baczem).

Np.

USE AdventureWorks

GO

SELECT \* FROM Production.Product

GO

– Dwa batch-e

**AND NOT →** zaprzeczenie określonego stwierdzenia czy warunku. W kodzie pierwszy warunek to po prostu NOT albo zwykłe stwierdzenie faktu - AND NOT występuje tu jako kolejny warunek.

Np.

USE AdventureWorks

GO

SELECT \* FROM Production.Product

WHERE

NOT ListPrice = 0

AND NOT ListPrice > 100

GO

– Pokazuje rekordy z ListPrice, które nie są równe 0 i nie są większe lub równe 100

**Różnica między AND i AND NOT** → AND w przykładowym AND ListPrice > 34.99 zwróci wartości większe od 34,99, natomiast AND NOT ListPrice < 34.99 zwróci wartości większe lub równe 34,99.

**Daty w klauzulach WHERE** → daty w klauzulach należy zapisywać w cudzysłowach.

Np.

SELECT \* FROM Production.Product

WHERE

SellStartDate > '2011.05.30'

GO

– Szuka dat po 2011.05.30

**BETWEEN** → Between szuka tego co mniejsze lub równe bądź większe lub równe. Wykorzystaj do liczb czy dat. Tutaj AND jest jakby przedłużeniem tej logiki BETWEEN czyli też szuka większe/mniejsze lub równe. Pamiętaj, że samo AND bez BETWEEN już się tak nie zachowuje. Składnia inna niż w poprzednich poleceniach w tym temacie - zaczyna się od nazwy kolumny. Kolumna BETWEEN wartość 1 AND wartość 2. Pamiętaj, że BETWEEN szuka zawsze elementów z włączeniem tego co wpisujesz czyli zawsze mniejsze/większe lub równe.

Np.

SELECT \* FROM Production.Product

WHERE

ListPrice BETWEEN 50 AND 100

GO

– Szuka wartości między 50, a 100 włączając w to 50 i 100

**IN** → szuka listy wartości, które są dopuszczalne w klauzuli WHERE. Wprowadź te wartości w nawiasie zwykłym. Składnia - tutaj również najpierw nazwa kolumny: Kolumna IN (warunek, warunek).

Np.

SELECT \* FROM Production.Product

WHERE

Color IN ('Black', 'Blue')

GO

**IN, a OR** → IN jest właściwie szybszym rozwiązaniem wobec wielokrotnego stosowania klauzuli OR. Jak na przykładzie poniżej IN robi to samo co wielokrotne OR.

Np.

USE AdventureWorks

SELECT \* FROM HumanResources.Employee

WHERE JobTitle IN ('Marketing Specialist','Control Specialist','Benefits Specialist','Accounts Receivable Specialist')

--WHERE JobTitle = 'Marketing Specialist' OR JobTitle = 'Control Specialist' OR JobTitle = 'Benefits Specialist' OR JobTitle = 'Accounts Receivable Specialist'

GO

**NOT IN** → odwrotność powyższego. Raczej unika się tego rozwiązania, bo SQL ma więcej pracy do wykonania - musi przejrzeć wszystkie rekordy. Raczej korzysta się z samego IN.

Np.

SELECT \* FROM Production.Product

WHERE

Color NOT IN ('Black', 'Blue')

GO

– Poszuka wszystkiego co nie zawiera tych kolorów.

**Kolejność AND i OR oraz nawiasy** → AND jest bardziej wiążące niż OR - jest to zasada jak w matematyce np. mnożenie jest silniej wiążące niż dodawanie. Do zapytań możesz też dodawać nawiasy jeśli masz wątpliwości jak zapytanie będzie interpretowane przez SQL - nawiasy zawsze mają pierwszeństwo.

Np.

Color = ‘Blue’ AND ListPrice > 100 OR Color = ‘Multi’ AND ListPrice < 50

– to to samo co:

(Color = ‘Blue’ AND ListPrice > 100) OR (Color = ‘Multi’ AND ListPrice < 50)

– Na obu przykładach SQL połączy kolor z ceną za pomocą AND, a OR będzie miało osobną rolę oddzielającą dwie możliwości. Nie ma problemu, że gdzieś to OR sprawi problem w stylu cena > 100 lub kolor = multi.

**Konieczność zastosowania nawiasów** → są takie sytuacje, gdzie chcę żeby OR i AND były oddzielone w specyficzny sposób. Wtedy korzystasz z nawiasów.

Np.

Color = 'Black' AND ListPrice > 100

OR

Color = 'Silver' AND ListPrice < 100

AND ProductLine = 'R'

GO

– Wyświetli 69 rekordów, bo tutaj OR oznacza ALBO srebrny, mniejszy niż 100 i z linii produktów R

(Color = 'Black' AND ListPrice > 100

OR

Color = 'Silver' AND ListPrice < 100)

AND ProductLine = 'R'

GO

– Wyświetli 30 rekordów, bo tutaj OR oznacza czarny i większy niż 100 srebrny ALBO mniejszy niż 100 i jednocześnie z linii produktów R

**17. Klauzula WHERE II**

**Operator LIKE** → Operator LIKE jest używany w klauzuli WHERE do wyszukiwania określonego wzorca w kolumnie. Istnieją dwa symbole wieloznaczne często używane w połączeniu z operatorem LIKE: Znak procentu (%) reprezentuje zero, jeden lub wiele znaków; Znak podkreślenia (\_) reprezentuje jeden, pojedynczy znak.

**Maski** → symbole wieloznaczne zastępujące określone znaki na określonych warunkach. Używane razem z LIKE. Wzór: Kolumna LIKE maska.

**Maska %** → % odpowiada za dowolną ilość dowolnych znaków. Można stawiać dwa, lub jeden po którejś stronie.

Np.

USE AdventureWorks

SELECT \* FROM Person.Address

WHERE PostalCode LIKE '9%'

GO

– Pokaże Wszystkie kody pocztowe zaczynające się od 9

**Maska []** → dopuszczanie występowania znaków, ale jako pojedyncze - to znaczy, że PostalCode LIKE ‘[89]47%’ będzie szukać albo 8 albo 9, a nie tylko 89.

Np.

USE AdventureWorks

SELECT \* FROM Person.Address

WHERE PostalCode LIKE '[123]%'

GO

– Pokaże wszystkie kody pocztowe zaczynające się od 1, 2 lub 3

**Żądanie żeby szukana wartość zaczynała się od jednej z cyfr** → to po prostu podanie wszystkich możliwych cyfr w nawiasie kwadratowym [0123456789]

**Określenie od czego szukane nie może się zacząć** → j/w, ale dodajesz daszek ^. Jeśli nie chcesz żeby zaczynało się od liczb to ‘[^0123456789]%’. Alternatywnie [^0-9]

Np.

USE AdventureWorks

SELECT \* FROM Person.Address

WHERE PostalCode LIKE '[^0-8]%'

GO

– Pokaże tylko kody pocztowe zaczynające się od 9 lub od liter

**Zakres, przykład** → ‘[^0123456789][0-9]%’ - szuka wartości, które zaczynają się literą, a po literze mają cyfrę od 0 do 9 czyli każdą cyfrę.

**Maska znak podkreślenia \_** → w tym miejscu ma się pojawić jeden dowolny znak. ‘6\_\_\_’ mają być 4 znaki, ale pierwszy z nich to 6.

Np.

USE AdventureWorks

SELECT \* FROM Person.Address

WHERE City LIKE '\_\_\_\_'

GO

– Szuka miast, które w nazwie mają 4 znaki.

**Usuwanie specjalnego znaczenia dla znaków w masce** → Sposoby:

1. Umieszczenie znaku w nawiasie kwadratowym: ‘%[%]%’
2. Wykorzystanie polecenie ESCAPE: LIKE ‘%/%%’ ESCAPE ‘/’

**ESCAPE** → określasz znak ucieczki, np. / i wykorzystujesz w połączeniu z ESCAPE jako instrukcję, że to co jest za tym znakiem ma stracić specjalne właściwości.

Np.

LIKE ‘%/%%’ ESCAPE ‘/’

**Klauzula OR i LIKE w jednym** → jeżeli chcę połączyć te dwie klauzule poszukując różnych możliwości, to pamiętaj, że każde OR zaczyna się od wyliczenia od nowa kolumny LIKE warunku.

Np.

USE AdventureWorks

SELECT \* FROM HumanResources.Employee

WHERE JobTitle LIKE '%Marketing%Specialist%' OR JobTitle LIKE '%Specialist%Marketing%'

GO

**18. Kolumny wyliczane**

**Definiowanie kolumn wyliczanych** → jeśli potrzebujesz wykonywać szybkie obliczenia lub łączyć w określony sposób kolumny, korzystasz z definiowania kolumn.

**CONCAT** → funkcja łącząca napisy ze sobą, ignorująca NULL co sprawia, że jest bardziej wiarygodna niż zwykły operator +. Przykład na dole.

**Definiowanie kolumn i wszystko (\*)** → możesz pracować na wszystkim (SELECT \*) po przecinku dodawać nowo-powstałe kolumny.

Np.

USE AdventureWorks

SELECT \*

    , YEAR(GetDate()) - YEAR(h.BirthDate) 'Age'

FROM HumanResources.Employee AS h

ORDER BY h.BirthDate DESC

**SELECT \*, nowa\_kolumna** → zwróć uwagę, że tworząc nową kolumnę, np. w celu posortowania w określony sposób, nadal stosujesz się do normalnych zasad, a co za tym idzie: gwiazdkę oznaczającą wszystko traktujesz jak normalny obiekt, jak jakąkolwiek kolumnę, więc dodając coś, po gwiazdce też będzie przecinek. To tak jakbyś chciał powiedzieć “Wybierz wszystko i…”

Np.

USE AdventureWorks

SELECT \*, YEAR(GetDate()) - YEAR(h.BirthDate) 'Age'

FROM HumanResources.Employee AS h

ORDER BY Age DESC

**Nadawanie nazwy kolumnie nowo utworzonej** → jeśli chcę utworzyć nazwę nowo nadawanej, wyliczanej kolumnie, to wystarczy, że na końcu SELECT napiszę w apostrofach jej pożądaną nazwę.

Np.

SELECT s.UnitPrice \* OrderQty 'Sales value'

FROM Sales.SalesOrderDetail AS s

GO

**Łączenie kolumn →** za pomocą operatora + lub za pomocą funkcji CONCAT. Pamiętaj, że + nie poradzi sobie, jeśli będą NULL-e. W takich przypadkach po prostu zwróci NULL.

Np.

USE AdventureWorks

SELECT 'Martial Status: ' + MaritalStatus + '; Gender: ' + Gender FROM HumanResources.Employee

GO

LUB

USE AdventureWorks

SELECT CONCAT ('Martial Status: ', MaritalStatus, '; Gender: ', Gender) FROM HumanResources.Employee

GO

**Liczenie wartości dynamicznie** → polega po prostu na prowadzeniu prostych obliczeń, tj. dodawaniu, odejmowaniu, mnożeniu, dzieleniu, etc. Wzór: nazwa kolumny i operator.

Np.

USE AdventureWorks

SELECT VacationHours / 8 FROM HumanResources.Employee

GO

**NULL** → to wartość nieokreślona.

**Umieszczanie apostrofu w apostrofie w SQL** → odbywa się za pomocą symbolu ucieczki, którym jest po prostu dodatkowy apostrof.

Np.

USE AdventureWorks

SELECT CONCAT('Day''s left:', VacationHours / 8, ', Hours left: ', VacationHours % 8) FROM HumanResources.Employee

GO

– Zwróci “Day’s left: 12”

**Obliczenia na liczbach** → korzystaj z nazw kolumn i operatorów obliczeniowych. Pamiętaj jednak, że np. standardowe dzielenie nie zwróci reszty i trzeba by dodać kolumnę z modulo lub wpleść to gdzieś w treść.

Np.

USE AdventureWorks

SELECT CONCAT('Day''s left:', VacationHours / 8, ', Hours left: ', VacationHours % 8) FROM HumanResources.Employee

GO

**YEAR** → funkcja zwracająca tylko rok z daty. Wzór: YEAR(data).

Np.

USE AdventureWorks

SELECT YEAR(BirthDate), YEAR(HireDate) FROM HumanResources.Employee

GO

**GetDate()** - funkcja zwróci datę dzisiejszą.

**Przykład z YEAR, CONCAT i GetDate →** Przykład.

Np.

USE AdventureWorks

SELECT CONCAT('Age: ', YEAR(GetDate()) - YEAR(BirthDate), ' and year''s employed: ',  YEAR(GetDate()) - YEAR(HireDate)) FROM HumanResources.Employee

WHERE (YEAR(GetDate()) - YEAR(HireDate)) % 5 = 0

LUB

USE AdventureWorks

SELECT CONCAT('Age: ', YEAR(GetDate()) - YEAR(BirthDate), ' and year''s employed: ',  YEAR(GetDate()) - YEAR(HireDate)) FROM HumanResources.Employee

WHERE (YEAR(GetDate()) - YEAR(HireDate)) IN (5, 10, 15, 20)

– Ten sam wynik co powyżej, ale tutaj zamiast z modulo skorzystałeś z operatora IN.

**SUM** → Funkcja sumująca wszystko z danej kolumny.

Np.

USE AdventureWorks

SELECT SUM (VacationHours) FROM HumanResources.Employee

GO

**19. Korzystanie z aliasów**

**No column name** → to komunikat, nazwa powstałem kolumny przy łączeniu kolumn. Można zmienić tą nazwę za pomocą aliasów.

**Aliasy** → AS (składnia) alias, czyli nazwa. Alias może być na początku wyrażenia albo później.

**AS** → nie trzeba używać, ale to dobra praktyka. Możesz użyć apostrofu albo nawiasu kwadratowego.

**Alias i WHERE** → Nie zadziała bo WHERE czyli filtr SQL uruchamia na początku. To co jest w zapytaniu za AS jeszcze nie istnieje. Żeby zadziałało powtórz w klauzuli WHERE ten zapis obliczeniowy który tworzy kolumnę z określoną nazwą za AS.

Np.

USE AdventureWorks

-- SELECT \* FROM Person.Person

SELECT CONCAT(FirstName, ' ', MiddleName) AS 'First and middle name'

FROM Person.Person

WHERE CONCAT(FirstName, ' ', MiddleName) LIKE 'A%'

GO

**Łączenie w jeden widok treść z kolumn z różnych tabel (połączonych kluczem)** → W SELECT wybierasz wszystkie kolumny które interesują Cię z obu tabel. FROM określa pierwszą tabelę. JOIN drugą tabelę. ON jako warunek dla łączenia, czyli, że inne kluczowe kolumny się ze sobą zgadzają.

SELECT

FirstName

,LastName

,PasswordSalt

FROM Person.Person

JOIN Person.Password

ON Person.Person.BusinessEntityID = Person.Password.BusinessEntityID

**Klauzula ON** → aby określić dowolne warunki lub określić kolumny do połączenia, używana jest klauzula ON. Używamy klauzuli ON, aby określić warunek łączenia. To pozwala nam określić warunki łączenia niezależnie od warunków wyszukiwania lub filtrowania w klauzuli WHERE.

**Skrócona notacja nazw tabel** → nadawanie im liter. Wzór = za nazwą tabeli wstaw znak lub ciąg znaków, które potem wykorzystasz jako odwołanie.

Np.

SELECT

FirstName

,LastName

,PasswordSalt

From Person.Person p

JOIN Person.Password e

ON p.BusinessEntityID = e.BusinessEntityID

**Błąd Ambigous column name** → wyświetla się kiedy SQL nie rozpoznaje w klauzuli ON którą tabelę ma wyświetlić, jeśli w dwóch różnych tabelach występuje kolumna o tej samej nazwie. W takim przypadku zaaliasuj w SELECT problematyczną kolumnę - one i tak są sobie równe.

Np.

SELECT

    LastName

    ,FirstName

    ,LoginID

    ,BusinessEntityID

FROM Person.Person p

JOIN HumanResources.Employee e

    ON p.BusinessEntityID = e.BusinessEntityID

GO

– Pokazuje błąd

SELECT

    LastName

    ,FirstName

    ,LoginID

    ,p.BusinessEntityID

FROM Person.Person p

JOIN HumanResources.Employee e

    ON p.BusinessEntityID = e.BusinessEntityID

GO

– p.BusinessEntityID zaliasowane

**Pisanie zapytania do tabeli, poprawna kolejność** → samo SELECT bez kolumn, FROM Tabela, aliasowanie i dopiero wtedy wymieniam kolumny w SELECT. Piszesz alias kropka i pokażą się możliwe do wyboru kolumny.

**Aliasy, kolumny i WHERE** → zaliasowanej kolumny nie można użyć w WHERE - tam trzeba powtórzyć obliczenie jeszcze raz.

Np.

USE AdventureWorks

SELECT CONCAT(FirstName, ' ', MiddleName) AS 'First and middle name'

FROM Person.Person

WHERE CONCAT(FirstName, ' ', MiddleName) LIKE 'A%'

GO

**20. Porządkowanie rekordów klauzula ORDER**

**Widok w SQL** → W języku SQL widok to wirtualna tabela oparta na zestawie wyników instrukcji SQL. Widok zawiera wiersze i kolumny, tak jak prawdziwa tabela. Pola w widoku to pola z jednej lub kilku rzeczywistych tabel w bazie danych. Do widoku można dodawać instrukcje i funkcje SQL oraz prezentować dane tak, jakby pochodziły z jednej tabeli.

**Unikanie sortowania** → sortowanie to czynność, której lepiej unikać jeśli to możliwe. To jedna z bardziej kosztownych operacji jakie wykonuje SQL Server. Sortowanie to czynność, którą można często wykonać w wersji prezentacyjnej, w aplikacji.

**Kolejność bez ORDER BY** → bez zapytania ORDER BY przy SELECT kolejność wyświetlania danych będzie losowa.

**ORDER BY** → instrukcja układająca w kolejności dane z tabeli. ORDER BY pojawia się po FROM. Docelowo sortowanie odbywa się w kolejności rosnącej, a nie malejącej, czyli od A do Z, od 0 do 100… Jeśli chcesz żeby kolejność się odwróciła na koniec instrukcji ORDER BY dodaj słowo DESC (jak descending).

Np.

USE AdventureWorks

SELECT

    s.CountryRegionName

    , s.StateProvinceName

    , s.City

    , s.Name

FROM [Sales].[vStoreWithAddresses] AS s

ORDER BY s.CountryRegionName

GO

– Od A do Z

USE AdventureWorks

SELECT

    s.CountryRegionName

    , s.StateProvinceName

    , s.City

    , s.Name

FROM [Sales].[vStoreWithAddresses] AS s

ORDER BY s.CountryRegionName DESC

GO

– Od Z do A

**ASC** → ascending. Słowo opcjonalne, określające, że ta kolumna am się sortować rosnąco.

**Więcej warunków sortowania i kolejność** → możesz sortować po większej ilości kolumn, według priorytetów od najważniejszego do najmniej ważnego. Po prostu wymieniaj je jedno po drugim. Możesz również w ramach sortowania wieloma kolumnami sortować albo malejąco albo rosnąco w zależności od potrzeb.

Np.

USE AdventureWorks

SELECT

    s.CountryRegionName

    , s.StateProvinceName

    , s.City

    , s.Name

FROM [Sales].[vStoreWithAddresses] AS s

ORDER BY s.CountryRegionName DESC, s.StateProvinceName, s.City DESC, s.Name

GO

**Szybkie sortowanie** → używanie nie nazw kolumn, ale numerów kolumn. Nie jest to rekomendowany sposób - ktoś mógłby dodać nową kolumnę i wtedy wszystko się wysypie. Lepiej używaj aliasów żeby pracować szybciej, one pod tym względem są bezpieczne.

Np.

USE AdventureWorks

SELECT

    h.DepartmentID

    , h.Name

    , h.GroupName

FROM HumanResources.Department AS h

ORDER BY h.DepartmentID DESC, h.Name, h.GroupName

GO

– Zwykły sposób

USE AdventureWorks

SELECT

    h.DepartmentID

    , h.Name

    , h. GroupName

FROM HumanResources.Department AS h

ORDER BY 1 DESC, 2, 3

GO

– Niebezpieczny sposób z numeracją

USE AdventureWorks

SELECT

    h.DepartmentID AS 'Dep'

    , h.Name AS 'Nam'

    , h. GroupName AS 'Gro'

FROM HumanResources.Department AS h

ORDER BY Dep DESC, Nam, Gro

GO

– Bezpieczny sposób z aliasami

**21. Typy danych - liczby całkowite**

**Typy danych** → znajdziesz je, m.in. w eksploratorze obiektów, w folderach. Typ danych określa jakie znaki i w jakich ilościach i o jakich funkcjach możesz użyć. Przykładowe, często używane typy danych:

nvarchar → pozwala zapisać 50 znaków, napisów zapisanych przy użyciu unicode;

datetime → pozwala na zapisanie daty i czasu;

int → liczba całkowita zakodowana przy użyciu 4 bajtów;

smallint → mniejsza wersja int, 2 bajty;

tinyint → jeszcze mniejsza wersja, 1 bajt;

bigint → 8 bajtów, wielkie wartości;

geography → zapamiętuje długości i szerokości geograficzne i pozwala na dokonywanie operacji na tym typie danych;

**DECLARE →** Deklarowanie zmiennych. Literał poprzedzać znakiem @. Robisz to poleceniem DECLARE. Wzór: DECLARE @literał TYP ZMIENNEJ. Declare jest potrzebne jako pierwsze do określenie jakiego typu to zmienna.

Np.

DECLARE @i INT

**SET** → tym poleceniem nadajesz wartość zmiennej. Wzór: SET @literał = wartość

Np.

DECLARE @i INT

SET @i=11

**PRINT** → polecenie wyświetlające wybrane wartości. Wzór: PRINT @literał.

Np.

DECLARE @i INT

SET @i=11

PRINT @i

**Działania matematyczne na zmiennych** → przy poleceniu SET można je wykonywać. Jeśli chcesz wykonać obliczenie szybko, to korzystaj ze znaku matematycznego i znaku równości, np. +=, -=, \*=.

Np.

DECLARE @i INT

SET @i=11

PRINT @i

SET @i = @i\*5

PRINT @i

SET @i = 10/3

PRINT @i

**Pobieranie wartości z tabeli, przypisanie zmiennej wartości z tabeli** → za pomocą SELECT FROM WHERE. Za Select określasz czemu równa ma być wartość, którą zadeklarowałeś. Najpierw oczywiście musisz zadeklarować o jaką zmienną ci chodzi. SELECT określa co wybierasz, czyli jaką zadeklarowaną zmienną, której zaraz przypiszesz wartość, FROM z jakiej tabeli WHERE musisz doprecyzować o którą dokładnie wartość chodzi, np. poprzez podanie klucza głównego.

Np.

USE AdventureWorks

DECLARE @i INT

SELECT @i=VacationHours FROM HumanResources.Employee WHERE BusinessEntityID=1

PRINT @i

**Wyliczenie wartości, a przypisanie z tabeli** → kiedy robisz to pierwsze korzystasz z SET a kiedy to drugie to z SELECT.

**Ograniczenia typów danych** → Int ma swoje ograniczenia wysokości liczb, które można zapisać na 4 bajtach. Smallint ma jeszcze większe ograniczenie. Używaj ich w zależności od potrzeb. Zwykły int to półśrodek, domyślnie stosowany, ale jak przewidujesz na pewno, niezmiennie określone wartości to korzystaj z pozostałych.

**DATALENGTH** → funkcja pozwalająca na sprawdzenie ilości pamięci konsumowanej przez poszczególne typy danych.

Np.

DECLARE @ti TINYINT = 0

DECLARE @si SMALLINT = 0

DECLARE @i INT = 0

DECLARE @bi BIGINT = 0

SELECT

DATALENGTH (@bi) AS [Bigint Size]

,DATALENGTH (@i) AS [Int Size]

,DATALENGTH (@si) AS [Smallint Size]

,DATALENGTH (@ti) AS [Tinyint Size]

**22. Typy danych - teksty**

**Typy napisowe lub znakowe** → 4 rodzaje, które występują w SQL.

char → znaki w bieżącej stronie kodowej. Zaleta: do zakodowania literki używam 1 bajt.

Np.

DECLARE @char CHAR(10) = ’POLAND’

**nchar** → zapamiętuje napisy zapisane przy użyciu UNICODE. W zamian potrzebne są 2 bajty.

Np.

DECLARE @nchar NCHAR(10) = N’POLAND’

**Kodowanie przy użyciu UNICODE** → przed deklarowaną zmienną wpisz “N”

Np.

DECLARE @nchar NCHAR(10) = N’POLAND’

**LEN** → funkcja określająca długość zmiennej. Wzór: LEN(zmienna).

Np.

USE AdventureWorks

DECLARE @char CHAR(10) = 'Leszek'

SELECT LEN(@char) AS 'DataLength'

**Określanie ile pamięci jest wykorzystane przez zmienne i długość napisów** → za pomocą funkcji LEN i DATALENGTH.

Np.

USE AdventureWorks

DECLARE @char CHAR(10) = 'Leszek'

DECLARE @nchar NCHAR(10) = N'Leszek'

SELECT @char AS 'Char', LEN(@char) AS 'Len', DATALENGTH(@char) AS 'Datalength'

SELECT @nchar AS 'NChar', LEN(@nchar) AS 'Len', DATALENGTH(@nchar) AS 'Datalength'

**Deklaracja bajtów, a koszta** →  W nawiasie DECLARE @char CHAR(10) = ’POLAND’ przekazujesz ile przewidujesz, że zmienna będzie miała znaków. Jeśli określisz, że 5, to w słowie “POLAND” SQL zapisze i będzie wyświetlać tylko “POLAN”. Pamiętaj jednak, że to co zadeklarujesz, to tyle będziesz poświęcać bajtów.

**VARCHAR** → typ zmiennej, która zajmuje tyle pamięci ile jest znaków w zmiennej. Zatem jeśli będzie zmienna zadeklarowana VARCHAR(10) = ‘Leszek’, to zmienna ta nie zajmie 10 bajtów, ale 6.

Np.

USE AdventureWorks

DECLARE @varchar VARCHAR(10) = 'Leszek'

SELECT @varchar AS 'VarChar', LEN(@varchar) AS 'Len', DATALENGTH(@varchar) AS 'Datalength'

**NVARCHAR** → podobnie jak w przypadku VARCHAR, ale tym razem jest to podwójna wartość, gdyż jest to zapisanie zmiennej w Unicode.Zatem jeśli będzie zmienna zadeklarowana VARCHAR(10) = ‘Leszek’, to zmienna ta nie zajmie 20 bajtów, ale 12.

Np.

USE AdventureWorks

DECLARE @nvarchar NVARCHAR(10) = N'Leszek'

SELECT @nvarchar AS 'VarNChar', LEN(@nvarchar) AS 'Len', DATALENGTH(@nvarchar) AS 'Datalength'

**CHAR, a VARCHAR** → generalnie lepiej jest korzystać z VARCHAR. Jest tak dlatego, że to oszczędność i czasu i miejsca. Tylko w przypadku, gdy jesteś na 100% pewny, że dana zmienna będzie miała zawsze określoną liczbę znaków, ma sens korzystanie z CHAR (np. zmienna PESEL, zawsze będzie miała 9 znaków).

**Lista wielkości typów danych** → Poniżej lista.

Bit: 1 bit

Tinyint: 1 bit

Smallint: 2 bity

Int: 4 bity

Bigint: 8 bitów

Char(n): n bitów

Varchar(n): n bitów + 2 bity

Nchar: 2 \* n bitów

Nvarchar: 2 \* n bitów

**23. Typy danych - data, liczba, prawda/fałsz**

**Daty można zapisywać za pomocą paru typów danych** → lista poniżej.

DATETIME: 8 bajtów. Funkcja GetDate(). Minisekundy.

DATETIME2: Funkcja SYSDATETIME(). Nanosekundy.

DATE: sama data. GetDate()

TIME: godzina. SYSDATETIME()

DATETIMEOFSET: zapamiętanie godziny z określeniem strefy czasowej. SYSDATETIME()

SMALLDATETIME: Data, godzina i sekundy. GetDate()

Np.

USE AdventureWorks

DECLARE @datetime DATETIME = GetDate()

DECLARE @datetime2 DATETIME2 = SYSDATETIME()

DECLARE @date DATE = GetDate()

DECLARE @time DATETIME = SYSDATETIME()

DECLARE @datetimeofset DATETIME = SYSDATETIME()

DECLARE @smalldate SMALLDATETIME = GetDate()

SELECT

@datetime AS 'DATETIME'

,@datetime2 AS 'DATETIME2'

,@date AS 'DATE'

,@time AS 'TIME'

,@datetimeofset AS 'DATETIMEOFSET'

,@smalldate AS 'SMALLDATETIME'

**Typy danych, a funkcje** → zwróć uwagę, że typy danych bywają podobne do funkcji, w sensie ich zapis. Pamiętaj jednak, że przy deklarowaniu zmiennych najpierw podajesz ich typ danych, a potem za pomocą funkcji nadajesz im wartość.

Np.

--DECLARE @d DATE = GetDate()

--DECLARE @d TIME = GetDate()

DECLARE @d SMALLDATETIME = GetDate()

PRINT @d

**Typy danych związane z liczbami zmiennoprzecinkowymi** → rekomenduje się korzystanie z Decimal aczkolwiek są różne. W przypadku zmiennej float nie deklaruje się cyfr i miejsc po przecinku tak dokładnie, podobnie jak w przypadku real. Nie używaj float i real w celu identyfikowania rekordów (nadawania kluczy). Typ money jest przestarzały.

**Decimal** → To typ danych, który pozwala na określenie ile maksymalnie cyfr chcesz zapamiętać w ramach danej liczby i ile spośród tych cyfr ma być po przecinku. Wzór: decimal(maksymalna ilość cyfr, ile cyfr po przecinku).

Np.

USE AdventureWorks

DECLARE @d decimal(11,2) = 123456789.1234

DECLARE @f float = 123456789.1234

DECLARE @r real = 123456789.1234

DECLARE @m money = 123456789.1234

SELECT

@d AS 'Decimal'

, @f AS 'Float'

, @r AS 'Real'

, @m AS 'Money'

**Typ logiczny** → w SQL typ logiczny wyraża się poprzez Bit. Bit pozwala na zachowanie wartości: 0 (FAŁSZ), 1 (PRAWDA), NULL (Wartość nie została zainicjowana).

Np.

USE AdventureWorks

DECLARE @b BIT = 1

SELECT @b AS 'BIT', DATALENGTH (@b) AS 'DATALENGHT'

**24. Funkcje znakowe**

**Funkcje znakowe** → funkcje w TSQL, czyli Transact SQL nie dają tak szerokich możliwości jak inne języki, ale można w TSQL wykonywać różne operacje. Funkcje należy pisać po przecinku.

**Funkcja CHARINDEX** → znajduje pozycje na której występuje określony znak. Przyjmuje argumenty: szukany znak, kolumna. Przykład poniżej.

Np.

SELECT

SUBSTRING(PropertyAddress, 1, CHARINDEX(',', PropertyAddress) -1),

SUBSTRING(PropertyAddress, CHARINDEX(',', PropertyAddress) +1, LEN(PropertyAddress))

FROM [Portfolio\_SQL Data Cleaning].dbo.NashvilleHousing

**Funkcja SUBSTRING** → Funkcja wycinająca treść z ciągu znaków. Argumenty: kolumna, pierwszy znak, ostatni znak). Przykład poniżej.

Np.

SELECT

SUBSTRING(PropertyAddress, 1, CHARINDEX(',', PropertyAddress) -1),

SUBSTRING(PropertyAddress, CHARINDEX(',', PropertyAddress) +1, LEN(PropertyAddress))

FROM [Portfolio\_SQL Data Cleaning].dbo.NashvilleHousing

**Zagnieżdżanie funkcji** -- w TSQL jest to możliwe. Przykład poniżej.

**Funkcja LEN** → Zwraca długość napisu w znakach. Przyjmuje jeden argument, kolumnę. Przykład poniżej.

**Funkcja UPPER** → zamienia wszystkie litery na wielkie. Argument: kolumna. Przykład poniżej.

**Funkcja LOWER** → zamienia wszystkie litery na małe. Argument: kolumna. Przykład poniżej.

Np.

USE AdventureWorks

SELECT LoginID

,UPPER(SUBSTRING(LoginID, (CHARINDEX('\',LoginID)+1),LEN(LoginID)))

FROM HumanResources.Employee

GO

**Funkcja FORMAT** → zwraca określony element w zadany sposób. Dwa sposoby wykorzystania funkcji, ale zwróć uwagę przedtem na prawidłowości. instrukcje w FORMAT są w cudzysłowach i formaty liczbowe automatycznie zaokrąglają podane wartości:

Sposób pierwszy: korzystasz z istniejących opcji. Argumenty: element do zmiany, stałą napisowa w jaki sposób ten element przedstawić, ustawienia lokalne komputera.

Sposób drugi: piszesz samodzielnie jak ma wyglądać format. Argumenty: element do zmiany, napisany format.

Np.

USE AdventureWorks

DECLARE @d DATETIME = GetDate()

SELECT FORMAT(@d, 'd', 'pl-PL')

SELECT FORMAT(@d, 'd', 'en-US')

SELECT FORMAT(@d, 'dd-MM-yyyy HH:mm')

SELECT FORMAT(123456789.3456, '0.0')

SELECT FORMAT(123456789.3456, '0.000')

**Funkcja LTRIM/RTRIM** → służą do usuwania białych znaków. LTRIM usuwa je z przodu, a RTRIM z tyłu napisu. Aby wykorzystać te dwie funkcje naraz zagnieźdź jedną w drugiej.

Np.

DECLARE @t VARCHAR(100) = '     some text     '

SELECT LTRIM(@t)

SELECT RTRIM(@t)

SELECT LTRIM(RTRIM(@t))

**Funkcja REPLACE** → zamienia napis wskazany jako drugi argument (pierwszym jest obiekt) na napis wskazany jako trzeci argument. Zamiana wielu napisów polega na zagnieżdżaniu funkcji.

Np.

SELECT PhoneNumber

,REPLACE(PhoneNumber, '-', '')

,REPLACE(REPLACE(REPLACE(REPLACE(PhoneNumber, '-', ''), '(', ''), ')', ''), ' ', '')

FROM Person.PersonPhone

**Funkcja REPLICATE** → funkcja powtórzy znak wskazany w pierwszym argumencie taką ilość razy jak podaje liczba przekazana przez drugi argument tej funkcji.

Np.

SELECT PostalCode

,REPLICATE('\_', 12-LEN(PostalCode))+PostalCode

FROM Person.Address

**Funkcja CONVERT** → pozwala na zmianę typu zmiennej. Pierwszy argument: nowy typ zmiennej, drugi: obiekt. Nie zapomnij określić nowej nazwy dla zmiennej

Np.

SELECT

TotalDue

,CONVERT(char(17), TotalDue) AS TotalDu

FROM Sales.SalesOrderHeader

**Źródło innych funkcji** → <https://www.techonthenet.com/sql_server/functions/index.php>

**25. Funkcje daty i czasu**

**Funkcje daty i czasu** → różne funkcje czasowe oddają różne typy zmiennych. GETDATE() oddaje typ datetime, SYSDATETIME() oddaje datetime2, a SYSDATETIMEOFFSET() oddaje czas ze strefą czasową.

**Deklarowanie zmiennej czasowej** → jeśli chcesz zadeklarować zmienną typu czas, musisz umieścić ją w cudzysłowiu, nieważne z jakiego formatu korzystasz.

Np.

SET LANGUAGE Polish

DECLARE @Dzień\_Urodzin DATETIME2

SET @Dzień\_Urodzin = '22-11-1992'

SELECT

DATENAME (dw, @Dzień\_Urodzin) 'Dzień moich Urodzin'

GO

**SWITCHOFFSET** → funkcja która pozwala zmienić strefę czasową podawaną w ramach SYSDATETIMEOFFSET(). W pierwszym argumencie podajesz datę, a w drugim w cudzysłowiu o ile chcesz zwiększyć lub zmniejszyć strefę czasową.

Np.

USE AdventureWorks

SELECT SYSDATETIMEOFFSET() 'Czas', SWITCHOFFSET (SYSDATETIMEOFFSET(), '+01:00') 'Czas w Paryżu'

**Funkcja YEAR** → funkcja pozwala na wyciąganie roku ze zmiennej typu data. Jako argument podajesz zmienną.

Np.

USE AdventureWorks

DECLARE @d DATETIME2 = GETDATE()

SELECT

YEAR(@d) 'Rok'

**Funkcja MONTH** → funkcja pozwala na wyciąganie miesiąca ze zmiennej typu data. Jako argument podajesz zmienną.

Np.

USE AdventureWorks

DECLARE @d DATETIME2 = GETDATE()

SELECT

MONTH(@d) 'Miesiąc'

**Funkcja DAY** → funkcja pozwala na wyciąganie dnia ze zmiennej typu data. Jako argument podajesz zmienną.

Np.

USE AdventureWorks

DECLARE @d DATETIME2 = GETDATE()

SELECT

DAY(@d) 'Dzień'

**Funkcja DATEPART** → funkcja ta pozwala na wyciąganie dowolnych części daty. W pierwszym argumencie podajesz skrót formatowania, które cię interesuje, a w drugim zmienną.

Np.

USE AdventureWorks

DECLARE @d DATETIME2 = GETDATE()

SELECT

DATEPART(YYYY,@d) 'Year',

DATEPART(M,@d) 'Month',

DATEPART(hh,@d) 'Hour',

DATEPART(n,@d) 'Minute',

DATEPART(dw,@d) 'WeekDay',

DATEPART(wk,@d) 'Week'

**Różnica między WeekDay, a Week** → weekday zwraca liczbę dnia tygodnia, a week, który to tydzień w danym roku.

**DATEFIRST i Dzień tygodnia w SQL** → zwróć uwagę, że domyślnie w SQL dzień tygodnia niedziela to pierwszy (1) dzień tygodnia, a sobota do ostatni (7). Poniższy przykład pisany w sobotę. Można to zmienić za pomocą funkcji SET DATEFIRST. Jeśli na początku kodu wpiszesz SET DATEFIRST 1 to pierwszym dniem tygodnia będzie poniedziałek. SET DATEFIRST 7 ustawi pierwszy dzień tygodnia jako niedzielę.

Np.

USE AdventureWorks

DECLARE @d DATETIME2 = GETDATE()

SELECT

DATEPART(dw,@d) 'WeekDay Sunday first'

SET DATEFIRST 1

SELECT

DATEPART(dw,@d) 'WeekDay Monday first'

**DATENAME** → Funkcja zmieniająca język zwracanych wartości typu data. Pisownia: SET LANGUAGE POLISH SELECT DATENAME(w pierwszym argumencie wpisuje się formatowanie daty, a w drugim konkretną datę)

Np.

USE AdventureWorks

SET LANGUAGE POLISH

SELECT DATENAME(M,'2023-09-02') 'Miesiąc'

GO

USE AdventureWorks

SET LANGUAGE FRENCH

SELECT DATENAME(M,'2023-09-02') 'Miesiąc'

GO

**DATEFORMAT** → funkcja, która odczytuje formatowanie daty w określony sposób. Wykorzystywane zwykle kiedy ktoś zapisuje datę w nietypowy sposób. Pisownia dla przykładu rok, miesiąc, dzień: SET DATEFORMAT ymd. Raz ustawiony format będzie działać dla całego zapytania.

Np.

USE AdventureWorks

SET DATEFORMAT dmy

DECLARE @Age DATETIME2 = '22.11.1992'

DECLARE @Today DATETIME2 = GetDate()

SELECT

DATEDIFF(dd, @Age, @Today) 'Dni życia',

DATEDIFF(dd, @Age, @Today)/365 'Lata życia'

GO

**DATEFROMPARTS** → funkcja, która złoży datę z podanych jej części w argumentach. Argumenty: rok, miesiąc, dzień.

Np.

USE AdventureWorks

SELECT DATEFROMPARTS(2024,12,31)

GO

**EOMONTH** → funkcja, która zwraca ostatni dzień danego miesiąca.

Np.

SELECT GetDate(), EOMONTH(GetDate())

**Wyciąganie pierwszego dnia miesiąca** → za pomocą DATEADD i DAY. Wynikiem DATEADD będzie data. Jak widać na przykładzie chodzi o odejmowanie dni danej daty, ale żeby zostawał jeden, pierwszy dzień miesiąca.

Np.

USE AdventureWorks

SELECT

DATEADD(dd, -(DAY(BirthDate)-1), BirthDate)

FROM HumanResources.Employee

**DATEADD** → funkcja dodaje do daty określony czas. Argumenty: interwał (czy dodajesz dni, miesiące, lata...), ile interwałów dodać, do jakiej daty chcesz to dodać czyli do jakiego obiektu czy kolumny). Zwróć uwagę, że wynikiem obliczeń będzie data, a nie int.

USE Adventureworks

SELECT h.OrderDate, DATEADD(d,2,OrderDate)

FROM Sales.SalesOrderHeader AS h

GO

**DATEADD i odejmowanie** → funkcji DATEADD używasz również żeby odejmować interwały czasowe od dat. Po prostu wykorzystujesz minus.

Np.

USE AdventureWorks

SELECT

h.BirthDate,

DATEADD(d,-32,BirthDate)

FROM HumanResources.Employee AS h

**DATEDIFF** → funkcja oblicza różnicę między dwoma datami. Argumenty: interwał (czy porównujesz jednostką dni, miesięcy, lat...), co porównuję, do czego). Uwaga: dla obliczania lat osób to może być YY albo YYYY.

Np.

USE Adventureworks

SELECT h.OrderDate, h.DueDate, DATEDIFF(d,OrderDate,DueDate)

FROM Sales.SalesOrderHeader AS h

GO

**Poszukiwanie dat w obrębie czasowym, ale względem jednej daty** → jeśli chcę wyświetlić tylko daty względem jednej daty (jakiś okres) to mogę skorzystać z BETWEEN i AND. Przykładowo chcę wyciągnąć rekordy dotyczące roku przed i roku po określonej dacie.

Np.

SELECT

\*

FROM HumanResources.Employee

WHERE BirthDate BETWEEN DATEADD(YEAR,-1,'1986-06-05') AND DATEADD(year,1,'1986-06-05')

GO

**Poszukiwanie dat w obrębie dwóch dat (pomiędzy) →** jeśli chcę wyciągać wszystkie daty większe od x i mniejsze od y wystarczy, że stosując się do formatu dat z bazy danych, np. w cudzysłowiu rok-miesiąc-dzień za pomocą znaków matematycznych i AND określę jaki zakres mnie interesuje.

Np.

USE AdventureWorks

SELECT

s.OrderDate

FROM Sales.SalesOrderHeader AS s

WHERE s.OrderDate >= '2012-01-01' AND s.OrderDate <= '2012-03-31'

**Deklarowanie/wyciąganie zmiennej typu data korzystając nie z przypisanej innej kolumny** → kiedy masz kolumnę z datą i chcesz wyciągnąć jakąś określoną wartość, musisz skorzystać z jakiejś kolumny obok, która jest przypisana do tego samego rekordu. Zadeklaruj zmienną, ale w miejscu przypisania wartości skorzystaj z SELECT/FROM/WHERE i naprowadź o jaki rekord chodzi.

Np.

DECLARE @BirthDate DATE

SELECT @BirthDate=BirthDate

FROM HumanResources.Employee

WHERE LoginID='adventure-works\diane1'

**Więcej funkcji** → <https://learn.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql?view=sql-server-ver16>

**26. Funkcje matematyczne**

**ABS** → funkcja zwraca wartości bezwzględne. ABS(wartość).

Np.

USE AdventureWorks

SELECT ABS(100), ABS(-100)

**CEILING** → funkcja zwraca największą bądź równą liczbę całkowitą wobec podanej wartości. Zwróć uwagę, że dla liczb dodatnich będzie to oznaczało wyższą cyfrę, a dla ujemnych niższą cyfrę, ale wyższą wartość, co jest zgodne z zasadami matematyki. Wzór: CEILING(wartość)

Np.

SELECT CEILING(1.5), CEILING(2.3), CEILING(-0.5), CEILING(-2.5)

**FLOOR** → funkcja zwraca najmniejszą bądź równą liczbę całkowitą wobec podanej wartości. Zwróć uwagę, że dla liczb dodatnich będzie to oznaczało niższą cyfrę i wartość, a dla ujemnych wyższą cyfrę, ale niższą wartość, co jest zgodne z zasadami matematyki. Wzór: FLOOR(wartość)

Np.

SELECT FLOOR(1.5), FLOOR(2.3), FLOOR(-0.5), FLOOR(-2.5)

**PI** → funkcja zwraca przybliżoną wartość liczby PI. Wzór: PI()

Np.

SELECT PI()

**POWER** → funkcja zwraca spotęgowaną wartość. Wzór: POWER(podstawa potęgi, wykładnik), czyli POWER(2,3) to dwa do potęgi trzeciej.

Np.

SELECT POWER(2,0), POWER(2,1), POWER(2,2), POWER(2,3)

**SQUARE** → funkcja pozwala na szybkie potęgowanie wartości do kwadratu. Alternatywa dla POWER, kiedy potęgujesz tylko do kwadratu. Wzór: SQUARE(wartość).

Np.

SELECT SQUARE(2), SQUARE(3), SQUARE(4)

**SQRT** → funkcja pierwiastkuje wartość. Wzór: SQRT(wartość).

Np.

SELECT SQRT(4), SQRT(9), SQRT(16)

**RAND** → funkcja zwraca wartość typu float z przedziału od 0 do 1, ale bez zera i jedynki, która jest wartością losową. Możesz wykonać jedna obliczenia, które zwrócą np. wartości losowe między 1, a 10, ale to będzie wymagało DECLARE i FLOOR. Przykłady poniżej.

Np.

SELECT RAND()

GO

DECLARE @random float = RAND()\*10

SELECT FLOOR(@random)

GO

**RAND, a poszukiwanie zakresu** → jeśli chcesz żeby funkcja RAND losowała liczby z danego zakresu, pomnóż RAND razy górę tego zakresu i umieść cały zapis w CEILING.

Np.

SELECT CEILING(RAND()\*49)

**ROUND** → funkcja zaokrąglająca. Przyjmuje dwa lub trzy argumenty. Wariant dwóch argumentów odda zaokrąglenie w górę, tj. ostatnia podana cyfra może zostać zaokrąglona w górę jeśli poprzednia jest w przedziale od 5 do 9, czyli program zachowa się zgodnie z zasadami matematyki. Możesz dodać trzeci argument, którym będize albo 0 (znowu zaokrąglenie) albo 1 (negacja zaokrąglenia, czyli oddanie ostatniej porządanej cyfry taka jaka jest, nawet jeśli za nią pojawia się 5-9. Wzór: ROUND(co zaokrąglasz, ile miejsc po przecinku ma zostać, opcjonalna zasada zaokrąglenia). W drugim argumencie ROUND można również podawać wartości ujemne, które mają za zadanie zaokrąglanie do jedności, dziesiątek, tysięcy, itd. -1 zaokrągli jedności, -2 dzieisątki, -3 setki, itd.

Np.

SELECT ROUND(12345.6789, 0), ROUND(12345.6789, 1), ROUND(12345.6789, 2), ROUND(12345.6789, 0, 1), ROUND(12345.6789, 1, 1), ROUND(12345.6789, 2, 1), ROUND(12345.6789, -1), ROUND(12345.6789, -2), ROUND(12345.6789, -3)

--Inne funkcje matematyczne: <https://learn.microsoft.com/en-us/sql/t-sql/functions/mathematical-functions-transact-sql?view=sql-server-ver16>

**27. Funkcje konwertujące**

**Wagi typów** → w przypadku kiedy chcę stworzyć napis składający się z różnych typów danych i np. napisać 'Total due equals' + TotalDue (gdzie TotalDue jest zmienną typu money) mogę natrafić na błąd. Błąd ten będzie wynikał z tego, że istnieją wagi typów. Wagi typów określają jakie typy plików są ważniejsze, a jakie mniej ważne (https://learn.microsoft.com/en-us/previous-versions/sql/sql-server-2012/ms190309(v=sql.110)?redirectedfrom=MSDN). W przykładzie próbuję połączyć typ money (13 pozycja na liście) z typem tekstowym (26-28 pozycja). Oznacza to, że konwersja działa tak, że tekst będzie konwertowany na money, co jest niemożliwe w tym przypadku.

**CAST** → funkcja konwertująca. Pozwala na konwertowanie typu danych na inny typ danych. CAST(jaki obiekt AS jaki typ danych).

Np.

USE AdventureWorks

SELECT TotalDue,

'Total due equals ' + CAST(TotalDue AS nvarchar(100))

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN (43661, 43671, 43681)

GO

**TRY\_CAST** → funkcja robi to samo co CAST, z tą różnicą, że jeśli funkcja napotka na błąd, to zamiast wyrzucać komunikat o błędzie, to zwróoci NULL.

Np.

DECLARE @txt CHAR(10)= '16/07/12'

SELECT TRY\_CAST(@txt AS DATE)

**CONVERT** → funkcja konwertująca. Pozwala na konwertowanie typu danych na inny typ danych. CONVERT(porządany typ danych, obiekt). Różnica między tym, a CAST polega na tym, że można w tej funkcji podać dodatkowy argument, który określa jak obiekt zostanie skonwertowany do napisu. W przykładzie data. Dla przykładu z datą zwróć uwagę, że number przypisany formatowaniu jest jednym z wielu dostępnych (https://learn.microsoft.com/en-us/sql/t-sql/functions/cast-and-convert-transact-sql?view=sql-server-ver16).

Np.

USE AdventureWorks

SELECT TotalDue,

'Total due equals ' + CONVERT(nvarchar(100), TotalDue)

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN (43661, 43671, 43681)

GO

DECLARE @d DATETIME = '2016-07-30'

SELECT

CONVERT(varchar(20), @d, 101) AS US

**TRY\_CONVERT** → funkcja robi to samo co CONVERT, z tą różnicą, że jeśli funkcja napotka na błąd, to zamiast wyrzucać komunikat o błędzie, to zwróoci NULL.

Np.

DECLARE @txt CHAR(10)= '16/07/12'

SELECT TRY\_CONVERT(DATE, @txt)

**Automatyczna konwersja** → konwersja z jednego typu danych na drugi jest dokonywana automatycznie kiedy obiekt na który konwertuje się jest wyżej na liście wagi typów niż obiekt z którego się konwertuje. W poniższym przykładzie konwersja z char na date.

Np.

DECLARE @txt CHAR(10) = '2016-07-12'

SELECT DAY(@txt)

GO

**PARSE** → kiedy sesja korzystania z SQL korzysta z ustawień dla danego kraju (np. ustawień amerykańskich) mogą pojawiać się błędy  gdy próbujesz, np. konwertować tekst na datę. Funkcja PARSE pozwala na chwilową zmianę ustawień. W przykładzie poniżej próbuję konwertować datę dzień/miesiąc/rok w ramach ustawień amerykańcikch, które pokazują najpierw miesiąc, a nie ma 16 miesiąca. Funkcja zmieni ustawienia na polskie. Wzór: zmienna, na jaki typ danych, jakie ustawienia. Funkcja ta pozwala nawet na identyfikację tekstu, np. "lipiec" na datę.

Np.

DECLARE @txt CHAR(10)= '16/07/12'

SELECT PARSE(@txt AS DATE USING 'pl-PL')

GO

DECLARE @txt CHAR(10)= '12 lipca 2016'

SELECT PARSE(@txt AS DATE USING 'pl-PL')

GO

**TRY\_PARSE** → funkcja robi to samo co PARSE, z tą różnicą, że jeśli funkcja napotka na błąd, to zamiast wyrzucać komunikat o błędzie, to zwróci NULL.

DECLARE @txt CHAR(10)= '12 lipca 2016'

SELECT TRY\_PARSE(@txt AS DATE USING 'de-DE')

GO

**28. Funkcje logiczne**

**Funkcje logiczne** → w Transact-SQL funkcje logiczne mają podejmować określone decyzje.

**IIF** → funkcja jeżeli. Przyjmuje trzy argumenty: test logiczny, co jeśli prawda, co jeśli fałsz. Poniższy przykład wyświetli dodatkową kolumnę z etykietami. Funkcję IIF można wykorzystać zarówno w klauzuli SELECT jak i w WHERE.

Np.

USE AdventureWorks

SELECT

[Name],

DaysToManufacture,

IIF(DaysToManufacture = 0, 'Can be done in one day', 'Needs longer time do produce')

FROM

Production.Product

**Wyliczanie kwartału z daty** → jeden sposób polega na sformatowaniu daty na pozostawienie tylko miesiąca i dodanie do tego miesiąca liczby 2 po czym podzielenie wyniku przez 3.

Np.

USE AdventureWorks

SELECT

SalesOrderID,

OrderDate,

(MONTH(OrderDate)+2)/3 AS 'Quarter'

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN(43659, 43764, 43872, 44116, 44366, 44521)

GO

**IFF zagnieżdżone w IFF** → czasami sytuacja wymaga, żeby poddać testowi więcej niż 2 możliwości, np. 3 i wtedy można zagnieździć IFF w IFF i można tak w nieskończoność. Dwa warunki są i jeśli nie to to może kolejny warunek i tak dalej...

Np.

USE AdventureWorks

SELECT

LoginID,hiredate,

IIF(DATEDIFF(year, HireDate, GETDATE())>10,'Old stager',IIF(DATEDIFF(year, HireDate, GETDATE())>8,'Veteran','Adept'))

FROM HumanResources.Employee

**CHOOSE** → zwraca element o określonym indeksie z listy wartości w programie SQL Server. Indeks jest wyrażeniem całkowitym, które reprezentuje indeks oparty na 1 na liście elementów następujących po nim. Jeśli podana wartość indeksu ma typ danych liczbowych inny niż int, wartość jest niejawnie konwertowana na liczbę całkowitą. Jeśli wartość indeksu przekracza granice tablicy wartości, funkcja CHOOSE zwraca wartość null.

val\_1 ... val\_n. Pierwszy argument funkcji musi być liczbą, a następne argumenty będą odpowiadały kolejności kolejnych wyników funkcji CHOOSE. Funkcja służy do wprowadzenia określonej zmiany na podstawie jakiegoś założenia.

Np.

USE AdventureWorks

SELECT

SalesOrderID,

OrderDate,

(MONTH(OrderDate)+2)/3 AS 'Quarter',

CHOOSE(

(MONTH(OrderDate)+2)/3,

'First quarter',

'Second quarter',

'Third quarter',

'Fourth quarter') AS 'Quarter name'

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN(43659, 43764, 43872, 44116, 44366, 44521)

GO

**CHOOSE i więcej opcji** → przykład pokazuje wykorzystanie funkcji CHOOSE przy określaniu dni tygodnia z numerycznej daty.

Np.

USE AdventureWorks

SET DATEFIRST 1

SELECT

s.SalesOrderID,

s.OrderDate,

CHOOSE(DATEPART(dw,s.OrderDate),'lunes','martes','miércoles','jueves','viernes','sábado','domingo') 'Day of week'

FROM

Sales.SalesOrderHeader s

**ISDATE** → funkcja logiczna sprawdzająca czy tekstowy ciąg znaków może zostać skonwertowany na datę. Zwróć uwagę, że w grę wchodzą tu ustawienia języka. W wersji amerykańskiej co innego będzie rozumiana jako data niż w polskiej. DATEFORMAT pokazuje, że przy zmianie formatu daty na inny tekst można zmienić na datę. TRUE to tu 1, FALSE to 0.

Np.

USE AdventureWorks

DECLARE @txt char(10) = '16/9/12'

SELECT ISDATE(@txt)

GO

USE AdventureWorks

DECLARE @txt char(10) = '16/9/12'

SET DATEFORMAT ymd

SELECT ISDATE(@txt)

GO

**ISNUMERIC** → funkcja sprawdzająca czy tekst można skonwertować na wartość numeryczną.

Np.

USE AdventureWorks

DECLARE @num\_1 char(10) = '123.123.123'

DECLARE @num\_2 char(10) = '123.123'

SELECT ISNUMERIC(@num\_1)

SELECT ISNUMERIC(@num\_2)

**Inne funkcje logiczne** → <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/logical-operators-transact-sql?view=sql-server-ver16>

**29. Wyrażenie CASE**

**Wyrażenie CASE** → instrukcja, która stanowi alternatywę dla CHOOSE. Pozwala na precyzyjniejsze określenie przypadków i przypisanych do nich warunków. Jej składnia to: CASE obliczenie dające wynik WHEN wynik\_1 THEN co ma się stać WHEN wynik\_2 THEN co innego ma się stać... ELSE co jeśli żaden z powyższych END AS 'Tytuł kolumny'. Zwróć uwagę, że CASE nie zamyka całej instrukcji w nawiasie, a między kolejnymi WHEN nie ma przecinków.

Np.

USE AdventureWorks

SELECT

SalesOrderID,

OrderDate,

MONTH((OrderDate)+2)/3 AS 'Quarter',

CASE(MONTH(OrderDate)+2)/3

WHEN 1 THEN 'First quarter'

WHEN 2 THEN 'Second quarter'

WHEN 3 THEN 'Third quarter'

WHEN 4 THEN 'Fourth quarter'

ELSE '???'

END AS 'Quarter name'

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN(43659, 43764, 43872, 44116, 44366, 44521)

GO

**CASE, a kolejność →** zwróć uwagę, że przy formowaniu CASE ma znaczenie kolejność określonych WHEN. Jeśli wszystkie odnoszą się do tego co ma się znaleźć w rekordzie, to spełnienie pierwszego warunku niweluje możliwość nadpisania przez kolejne, które też byłyb spełnione, itd.

Np.

USE AdventureWorks

SELECT ProductID, [Name],

CASE

WHEN [Weight] IS NOT NULL THEN CAST([Weight] AS VARCHAR)

WHEN Size IS NOT NULL THEN Size

WHEN Color IS NOT NULL THEN Color

ELSE 'Call for more info'

END AS 'Description',

[Weight], Size, Color

FROM Production.Product

**CASE alternatywna składnia** → jeśli działasz na bardziej skomplikowanych warunkach logicznych, możesz skorzystać z innej składni CASE. Polega ona na tworzeniu warunków logicznych za kolejnymi instrukcjami WHEN. Warunki te muszą oddawać wynik TRUE albo FALSE i zdarzenie zdarzy się jeśli tylko warunek zostanie spełniony, czyli TRUE. Składnia: CASE WHEN warunek logiczny THEN wynik... ELSE... END... AS. Zwróć uwagę w poprzednim przykładzie, że kiedy chciałeś przypisać coś nowego dla wartości NULL to nie napisałeś WHEN x = NULL tylko x IS NULL. Ponadto zauważ, że ostatnie ELSE nie ma nic przypisane i oznacza to, że inne przypadki nie wynienione w instrukcji CASE pozostaną bez zmian, nie są brane pod uwagę.

Np.

USE AdventureWorks

SELECT

p.ProductID,

p.[Name],

p.Color,

CASE

WHEN p.Color IS NULL THEN 'Not painted'

WHEN p.Color = 'Black' THEN 'Grey'

WHEN p.Color = 'Blue' THEN 'Navy'

WHEN p.Color = 'Red' THEN 'Amaranth'

ELSE p.Color

END AS 'New color'

FROM

Production.Product p

GO

**CASE, WHEN i OR** → jeśli chcesz żeby wynik odnosił się do dwóch lub więcej warunków, zastosuj klauzulę OR, ale powtórz za każdym razem całą składnię warunku logicznego.

Np.

USE AdventureWorks

SELECT

p.PhoneNumberTypeID,

CASE

WHEN [Name] = 'Cell' THEN 'Mobile phone'

WHEN [Name] = 'Home' THEN 'Stationary'

WHEN [Name] = 'Home' OR [Name] = 'Work' THEN 'Stationary'

ELSE 'Other'

END AS 'New name'

FROM Person.PhoneNumberType p

GO

**Dokumentacja funkcji** → <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/case-transact-sql?view=sql-server-ver16>

**30. Funkcje agregujące**

**Funkcje agregujące** → funkcja agregująca to funkcja, która pracuje w oparciu nie o jeden rekord, wyliczająca coś z danych jednego rekordu, ale pracująca na zbiorze rekordów.

**COUNT** → funkcja liczy ilość rekordów w danej kolumnie. Kiedy interesuje cię po prostu ilość rekordów to w argumencie funkcji umieść po prostu gwiazdkę - tak jest bezpieczniej. Jest to też istotne ponieważ funkcja COUNT nie bierze w obliczeniach pod uwagę NULL. Jak widać na poniższym przykładzie zliczanie względem name i gwiazdki daje ten sam wynik, bo każdy rekord ma nazwę. Nie każdy natomiast na kolor.

Np.

USE AdventureWorks

SELECT COUNT([name]) AS 'Number of records with names' FROM Production.Product

SELECT COUNT(\*) AS 'Number of all records' FROM Production.Product

SELECT COUNT(Color) AS 'Number of all records with color' FROM Production.Product

**AVG** → funkcja agregująca obliczająca średnią. W argumencie wpisujesz kolumnę dla której mają być prowadzone obliczenia. Pamiętaj, że rekordy gdzie są NULL-e nie będą brane pod uwagę w obliczeniach.

Np.

SELECT AVG(ListPrice) AS 'Average' FROM Production.Product

**MIN** → funkcja agregująca obliczająca mininalną wartość. W argumencie wpisujesz kolumnę dla której mają być prowadzone obliczenia. Pamiętaj, że rekordy gdzie są NULL-e nie będą brane pod uwagę w obliczeniach.

Np.

SELECT MIN(ListPrice) AS 'Minimum' FROM Production.Product

**MAX** → funkcja agregująca obliczająca maksymalną wartość. W argumencie wpisujesz kolumnę dla której mają być prowadzone obliczenia. Pamiętaj, że rekordy gdzie są NULL-e nie będą brane pod uwagę w obliczeniach.

Np.

SELECT MAX(ListPrice) AS 'Maximum' FROM Production.Product

**SUM** → funkcja agregująca obliczająca sumę wartości. W argumencie wpisujesz kolumnę dla której mają być prowadzone obliczenia. Pamiętaj, że rekordy gdzie są NULL-e nie będą brane pod uwagę w obliczeniach.

Np.

SELECT SUM(ListPrice) AS 'Sum' FROM Production.Product

**GROUP BY** → funkcja agregująca, która grupuje wszystkie rekordy za pomocą określonej kategorii. Chodzi o to, że masz dane w jednej kolumnie i są one przypisane do węższych kategorii w innej kolumnie (np. kolumna z produktami typu farby, oleje, gwoździe są przypisane do kategorii budowlanka) i chcesz zobaczyć ile obiektów znajduje się w określonych kategoriach. Żeby tego dokonać wybierasz (SELECT) kolumnę z kategoriami i liczysz (COUNT) wszystkie (\*) obiekty w bazie. Musisz jeszcze pod FROM umieścić klauzulę GROUP BY kategorie. Wynik możesz posortować.

Np.

USE Adventureworks

SELECT

p.ProductSubcategoryID,

COUNT(\*) AS 'Number of products in subcategory'

FROM Production.Product AS p

GROUP BY p.ProductSubcategoryID

ORDER BY p.ProductSubcategoryID DESC

**Skomplikowana wersja GROUP BY** → w poniższym przykładzie wykorzystuje się poznane już elementy polecenia SELECT. Określoną podkategorię (WHERE), określone minimum dla obiektów (MIN) - czyli, że chcę zobaczyć najmniejsze wartości dla danej kategorii.

Np.

SELECT

p.ProductSubcategoryID AS 'Subcategory'

,MIN(ListPrice) AS 'MinPriceInSubCategory'

FROM Production.Product AS p

WHERE ProductSubcategoryID IS NOT NULL

GROUP BY p.ProductSubcategoryID

ORDER BY MinPriceInSubCategory DESC

**GROUP BY zasada działania** → zwróć uwagę, że GROUP BY działa tak, że grupuje poszczególne kolumny za pomocą określonego klucza. Kolumny te występują pod funkcją SELECT oraz w zapisie samego GROUP BY. Na końcu SELECT i tylko tam (a już nie w funkcji GROUP BY) znajduje się natomiast właśnie ten klucz na podstawie, którego grupowane są dane z wymienionych kolumn.

Np.

USE AdventureWorks

SELECT

MONTH(s.OrderDate) AS 'Month',

s.SalesPersonID,

s.TerritoryID,

SUM(s.SubTotal) AS 'Sum'

FROM Sales.SalesOrderHeader AS s

WHERE s.OrderDate >= '2012-01-01' AND s.OrderDate <= '2012-03-31' AND s.SalesPersonID IS NOT NULL AND s.TerritoryID IS NOT NULL

GROUP BY MONTH(s.OrderDate),s.SalesPersonID,s.TerritoryID

**HAVING\_1** → wyrażenie filtrujące ilość rekordów które zostaną zwrócone patrząc na wartość, która została wyliczona przez funkcję agregującą. Jest to żądanie żeby wyświetlać te obiekty, które spełniają określony warunek. Jest to jakby drugie filtrowanie po WHERE (które filtruje po określonych rekordach) - HAVING filtruje po grupach utworzonych w funkcji agregującej. Jest tylko jeden problem: w klauzuli HAVING nie możesz korzystać z aliasów.

Np.

SELECT

p.ProductSubcategoryID AS 'Subcategory'

,MIN(ListPrice) AS 'MinPriceInSubCategory'

FROM Production.Product AS p

WHERE ProductSubcategoryID IS NOT NULL

GROUP BY p.ProductSubcategoryID

HAVING MIN(ListPrice) > 500

ORDER BY MinPriceInSubCategory DESC

**Logiczny porządek wykonywania obliczeń w Transact-SQL** → poniżej znajduje się lista z przypisaną kolejnością jak należy budować zapytanie aby korzystać z aliasów. Chodzi o to jakie polecenia są wykonywane w jakiej kolejności.

/\*

1. FROM

2. ON

3. JOIN

4. WHERE

5. GROUP BY

6. WITH CUBE/ROLLUP

7. HAVING

8. SELECT

9. DISTINCT

10. ORDER BY

11. TOP

12. OFFSET/FETCH

\*/

**STDEV** → funkcja zwraca statystyczne odchylenie standardowe wszystkich wartości w określonym wyrażeniu.

Np.

USE AdventureWorks

SELECT

STDEV(p.ListPrice)

FROM

Production.Product p

WHERE p.ProductSubcategoryID = 14

**HAVING\_2** → funkcja określa warunek wyszukiwania grupy lub agregatu. HAVING można używać tylko z instrukcją SELECT. HAVING jest zwykle używany z klauzulą GROUP BY. Gdy nie jest używana opcja GROUP BY, istnieje niejawna pojedyncza, zagregowana grupa.

Np.

USE AdventureWorks

SELECT

s.SalesPersonID,

SUM(SubTotal) AS 'TotalSales'

FROM

Sales.SalesOrderHeader s

WHERE YEAR(OrderDate) = 2012

GROUP BY s.SalesPersonID

HAVING SUM(SubTotal) > 100000

ORDER BY 'TotalSales' DESC

**Wykorzystanie paru funkcji naraz i grupowanie** → jak w przykładzie poniżej, jest to możliwe.

Np.

USE AdventureWorks

SELECT AVG(p.ListPrice - p.StandardCost) AS 'Average', MIN(p.ListPrice - p.StandardCost) AS 'Minimal', MAX(p.ListPrice - p.StandardCost) AS 'Maximal'

FROM Production.Product p

WHERE p.ListPrice > 0 OR p.StandardCost > 0

GROUP BY p.ProductSubcategoryID

ORDER BY 'Average' DESC

**GROUP BY dla wielu kolumn** → jeśli chcę agregować, grupować rekordy za pomocą jakiegoś zewnętrznego obliczenia (np. sumy) muszę w GROUP BY podać wszystkie kolumny, poza tą zewnętrzną, które przywołuję.

Np.

USE AdventureWorks

SELECT

MONTH(s.OrderDate) AS 'Month',

s.SalesPersonID,

s.TerritoryID,

SUM(s.SubTotal) AS 'Sum'

FROM Sales.SalesOrderHeader AS s

WHERE s.OrderDate >= '2012-01-01' AND s.OrderDate <= '2012-03-31' AND s.SalesPersonID IS NOT NULL AND s.TerritoryID IS NOT NULL

GROUP BY MONTH(s.OrderDate),s.SalesPersonID,s.TerritoryID

**31. Null i funkcje pracujące z NULL**

**NULL i funkcje pracujące z NULL** → NULL to wartość nieokreślona, nie można więc powiedzieć czy jest większe, mniejsze czy równe zero. Jednocześnie nie można użyć NULL w standardowy sposób w klauzuli WHERE, bo znowu to wartość nieokreślona - WHERE zwraca wartości dla których jest TRUE, a nie FALSE czy jak w przypadku NULL - UNKNOWN. Dalej - nie można również w WHERE wyszukać wartości różnych od NULL (<>) bo to wartość nieokreślona, więc program nie wie czego szuka.

**Konstrukcja do pracy z NULL (IS)** →  należy wykorzystać IS zamiast znaków matematycznych takich jak znak równości.

Np.

USE AdventureWorks

SELECT SpecialOfferID, DiscountPct, Category, MinQty, MaxQty

FROM Sales.SpecialOffer

WHERE MaxQty IS NULL

**IS NOT NULL** → przeciwieństwo IS, czyli sprawdza co nie ma NULL.

Np.

USE AdventureWorks

SELECT SpecialOfferID, DiscountPct, Category, MinQty, MaxQty

FROM Sales.SpecialOffer

WHERE MaxQty IS NOT NULL

**ANSI\_NULLS** → instrukcja uruchamiająca możliwość wykorzystania znaków matematycznych do pracy z NULL. Porównania do wartości NULL powinny już działać. Lepiej jednak korzystać ze składni IS.

Np.

SET ANSI\_NULLS OFF

USE AdventureWorks

SELECT SpecialOfferID, DiscountPct, Category, MinQty, MaxQty

FROM Sales.SpecialOffer

WHERE MaxQty = NULL

**Działania na NULL** → wynikiem działania na NULL będzie NULL.

**NULL, a ORDER BY** → NULL zostało skonstruowane w taki sposób żeby NULL stanowiło najmniejszą wartość.

USE AdventureWorks

SELECT SpecialOfferID, DiscountPct, Category, MinQty, MaxQty

FROM Sales.SpecialOffer

ORDER BY MaxQty DESC

**ISNULL i ukrywanie wartości NULL** → kiedy chcę wartość NULL ukryć lub zastąpić czymś innym korzystam z funkcji ISNULL. Pierwszy argument to kolumna, drugi to obiekt który podmieniam w miejsce NULL). Przy "ukrywaniu" podmień na stosunkowo dużą liczbę, która będzie zrozumiała dla każdego użytkownika.

USE AdventureWorks

SELECT SpecialOfferID, DiscountPct, Category, MinQty, ISNULL(MaxQty, 9999999)

FROM Sales.SpecialOffer

**COALESCE** → funkcja robiąca to samo co ISNULL i więcej. Do tej funkcji można podać więcej niż jedną kolumnę po czym wykonywany jest test. Jeśli pierwsza podana kolumna ma NULL, to bierze kolejną wartość, itd.

Np.

USE AdventureWorks

SELECT ProductID, [Name], COALESCE(CAST([Weight] AS VARCHAR), Size, Color, 'Call for more info') AS 'Description', [Weight], Size, Color

FROM Production.Product

**ISNULL złożone z efektem COALESCE** → teoretycznie możliwe jest rozpisanie ISNULL tak, żeby miało funkcjonalność COALESCE, ale wymaga to zagnieżdżania wielu funkcji ISNULL w sobie.

Np.

USE AdventureWorks

SELECT ProductID, [Name], ISNULL(CAST([Weight] AS VARCHAR), ISNULL(Size, ISNULL(Color, 'Call for more info'))) AS 'Description', [Weight], Size, Color

FROM Production.Product

**CASE jako alternatywa dla ISNULL i COALESCE** → teoretycznie też można w ogóle zrezygnować z ISNULL i COALESCE. Można wykorzystać CASE i IS NOT NULL.

Np.

USE AdventureWorks

SELECT ProductID, [Name],

CASE

WHEN [Weight] IS NOT NULL THEN CAST([Weight] AS VARCHAR)

WHEN Size IS NOT NULL THEN Size

WHEN Color IS NOT NULL THEN Color

ELSE 'Call for more info'

END AS 'Description',

[Weight], Size, Color

FROM Production.Product

**CREATE TABLE # →** funkcja tworząca tymczasową tabelę. W argumentach wpisz nazwy kolumn, przy czym na początku daj Id INT IDENTITY PRIMARY KEY żeby określić klucz podstawowy). Jeśli dodasz na początku znak hash to będzie to oznaczało przygotowanie tabeli tymczasowej.

Np.

CREATE TABLE #ContactHistory(Id INT IDENTITY PRIMARY KEY, CustomerId INT, OldContact NVARCHAR(50), NewContact NVARCHAR(50), OldAddress NVARCHAR(50), NewAddress NVARCHAR(50))

SELECT \* FROM #ContactHistory

**NULLIF** → funkcja służy do zwracania wartości NULL w przypadku gdy oba argumenty podane są takie same. Działa tak, że jeśli oba argumenty są takie same, zwróci NULL. Jeśli są różne to funkcja zwróci wartość pierwszego argumentu. Służy to chociażby do aktualizacji danych gdy chcesz żeby nowa kolumna nie zwracała danych jeśli te pozostają niezmienne, tylko żeby pokazało się NULL. Dodaj funkcję ISNULL jeśli chcesz żeby użytkownik widział jakiś komunikat baridzej jasny niż samo NULL (np. nawet samo puste "").

Np.

CREATE TABLE #ContactHistory(Id INT IDENTITY PRIMARY KEY, CustomerId INT, OldContact NVARCHAR(50), NewContact NVARCHAR(50), OldAddress NVARCHAR(50), NewAddress NVARCHAR(50))

INSERT INTO #ContactHistory VALUES(101, NULL, 'Kevin Brown', NULL, '7484 Roundtree Drive')

INSERT INTO #ContactHistory VALUES(101, 'Kevin Brown', 'Kevin Brown', '7484 Roundtree Drive', '6387 Scenic Avenue')

INSERT INTO #ContactHistory VALUES(101, 'Kevin Brown', 'Kevin Brown', '6387 Scenic Avenue', '6387 Scenic Avenue')

INSERT INTO #ContactHistory VALUES(101, 'David Lugo', 'Andrew Hill', '6387 Scenic Avenue', '6387 Scenic Avenue')

SELECT \* FROM #ContactHistory

SELECT Id, CustomerID, ISNULL(NULLIF(NewContact, OldContact),'') AS 'Change in contact', ISNULL(NULLIF(NewAddress, OldAddress),'') AS 'Change in address'

FROM #ContactHistory

**DROP TABLE** → funkcja usuwająca tabelę. Na przykładzie widać usunięcie tabeli tymczasowej, która teoretycznie sama zostałaby usunięta bo jest tymczasowa.

Np.

CREATE TABLE #ContactHistory(Id INT IDENTITY PRIMARY KEY, CustomerId INT, OldContact NVARCHAR(50), NewContact NVARCHAR(50), OldAddress NVARCHAR(50), NewAddress NVARCHAR(50))

DROP TABLE #ContactHistory

**32. SELECT DISTINCT i SELECT TOP**

**SELECT DISTINCT** → polecenie pozwala na wyciągnięcie tylko unikalnych rekordów z kolumny lub unikalnych par rekordów z dwóch kolumn, itd. Wzór SELECT DISTINCT kolumna\_1 FROM tabela\_1

Np.

USE AdventureWorks

SELECT DISTINCT Color, Class FROM Production.Product

**SELECT TOP** → polecenie pozwala na wyciąganie n-tej liczby obiektów spełniających określone warunki.

Np.

USE AdventureWorks

SELECT TOP(3)

[Name], ListPrice

FROM Production.Product

ORDER BY ListPrice DESC

**SELECT TOP WITH TIES** → polecenie pozwala na wyciąganie n-tej liczby obiektów spełniających określone warunki, ale jeśli istnieją obiekty w kolumnach lub pary, etc., które są takie same to mimo przekroczenia ilości pytanyc obiektów w TOP() podane zostaną wszystkie takie same obiekty/połączenia.

Np.

USE AdventureWorks

SELECT TOP(3) WITH TIES

[Name], ListPrice

FROM Production.Product

ORDER BY ListPrice DESC

**SELECT TOP PERCENT** → polecenie wyświetla rekordy spełniające podane warunki i przyjamniej w przypadku wartości liczbowych stanowi x procent, który podałem. Można łączyć z WITH TIES i wtedy powtórzony zostanie każdy rekord, którego choć jedna wersja uwzględniona by była bez WITH TIES.

Np.

USE AdventureWorks

SELECT TOP(1) PERCENT

[Name], ListPrice

FROM Production.Product

ORDER BY ListPrice DESC

**TABLESAMPLE** → polecenie wybiera losową część rekordów z tabeli, celem zwykle zapoznania się z jej zawartością. Możesz wybierać określoną ilość losowych rekortów podając procenty lub wiersze. Zwróć uwagę, że czasem SQL zwracać będzie pustą tabelę, więc możesz odświeżać parę razy.

USE AdventureWorks

SELECT

[Name], ListPrice

FROM Production.Product TABLESAMPLE(1 PERCENT)

USE AdventureWorks

SELECT

[Name], ListPrice

FROM Production.Product TABLESAMPLE(500 ROWS)

**33. GROUP BY, ROLLUP i CUBE**

**ROLLUP jako dodatek do GROUP BY** → pamiętaj, że GROUP BY łączy wszystkie rekordy mające takie same wartości i agreguje inne wartości, które mnie interesują. Na poniższym przykładzie widać jak chcę poznać średnią cenę butów dla każdej kombinacji ich koloru, rozmiaru i klasy. Dopisana dodatkowo klauzula ROLL UP pozwala na obliczenie interesującej mnie wartości dla każdej możliwej kombinacji wybranych w argumentach ROLL UP. Zwróć uwagę, że mimo, że w klauzuli WHERE pojawia się warunek nie podawania NULL to NULL występuje w rekordach wynikowych. Jest tak, ponieważ NULL w tych wynikach oznacza, że kombinacje dla dwóch pozostałych wartości z kolumn niezależnie od kolumny NULL wynoszą określoną wartość. W klauzuli ROLL UP pojawia się uogólniony wynik (w przykładzie) zawsze od prawej strony do lewej (od Class do Color) i dlatego nie widzisz uogólnionego wyniku np. dla Class, ale dla Color już tak. Do uogólniania od lewej do prawej jest inny operator: CUBE.

Np.

USE AdventureWorks

SELECT

p.Color,

p.Size,

p.Class,

AVG(p.ListPrice) AS AveragePrice

FROM Production.Product AS p

WHERE p.Color IS NOT NULL AND p.Size IS NOT NULL AND p.Class IS NOT NULL

GROUP BY ROLLUP (p.Color, p.Size, p.Class)

**CUBE** → operator robiący to co ROLL UP, ale od lewej do prawej strony. Operator ten analizuje jakby wszystkie strony jednocześnie, wszystkie możliwe połączenia, dlatego nazywa się kostką.

Np.

USE AdventureWorks

SELECT

p.Color,

p.Size,

p.Class,

AVG(p.ListPrice) AS AveragePrice

FROM Production.Product AS p

WHERE p.Color IS NOT NULL AND p.Size IS NOT NULL AND p.Class IS NOT NULL

GROUP BY CUBE (p.Color, p.Size, p.Class)

**GROUPING SETS** → operator będący rozwiązaniem pomiędzy ROLLUP, a CUBE. W argumentach wybierasz jakie grupy lub pojedyncze kolumny chcesz łączyć do analizy. Używaj kiedy dokładnie wiesz jakie wartości w zestawieniu są ci potrzebne.

Np.

USE AdventureWorks

SELECT

p.Color,

p.Size,

p.Class,

AVG(p.ListPrice) AS AveragePrice

FROM Production.Product AS p

WHERE p.Color IS NOT NULL AND p.Size IS NOT NULL AND p.Class IS NOT NULL

GROUP BY GROUPING SETS (p.Color, (p.Color, p.Size, P.Class), p.Class)

**Wizualna różnica między ROLLUP, a CUBE** → w rezultacie poniższego ROLL UP otrzymamy sumę pola D na następujących poziomach: ABC, AB, A. Natomiast zapytanie CUBE pozwoli nam otrzymać sumę na następujących poziomach: ABC, AB, AC, A, BC, B, C.

Np.

SELECT

     A,

     B,

     C,

     sum(D)

FROM

     Table\_name

GROUP BY

     ROLLUP/CUBE(A,B,C)

**34. Funkcja GROUPING\_ID**

**FUNKCJA GROUPING\_ID** → funkcja, która zwraca informację czy przy grupowaniu dana kolumna została zagregowana. Jeśli została zagregowana, czyli wykonano na niej zgrupowanie wartości to zwróci 1, a jeśli pozostała nietknięta, a zatem oddaje tylko określoną wartość z kolumny, to zwróci 0. Może służyć do odpowiedniego interpretowania agregowania danych, gdzie każda kolumna zwraca NULL w więcej niż jednym przypadku. Jak na przykładzie poniżej: w pierwszym wierszu widać średnią cenę dla każdego koloru, każdego rozmiaru, ale tylko dla produktów z nieokreśloną klasą. Natomiast wiersz piąty podaje dane dla każdego koloru, ale produktów z nieokreślonym rozmiarem i klasą.

Np.

USE AdventureWorks

SELECT

p.Color, GROUPING\_ID(p.Color) AS 'Is color aggregated',

p.Size, GROUPING\_ID(p.Size) AS 'Is size aggregated',

p.Class, GROUPING\_ID(p.Class) AS 'Is class aggregated',

AVG(p.ListPrice) AS AveragePrice

FROM Production.Product AS p

GROUP BY GROUPING SETS ((p.Color), (p.Size), (P.Class))

**GROUPING\_ID i wiele argumentów** → jeśli podasz w tej funkcji więcej niż jeden argument, to dostaniesz sumę potęg liczby 2, patrząc od pierwszego wiersza. Zatem jak na przykładzie: w pierwszym wierszu wynik to 6 ponieważ dwie kolumny zostały zagregowane, pierwsza i druga, każda zwraca 1 - są zatem dwie jedynki, więc dwa do potęgi pierwszej plus dwa do potęgi drugiej = 6. Gdyby ostatnia kolumna zwracała 1 to mielibyśmy jeszcze plus dwa do potęgi zero czyli 4+2+1 = 7. Wynik ten może być bardziej przejrzysty dla aplikacji, człowiek może woleć osobne kolumny jak dotychczas. Pamiętaj dwa do potęgi w zależności ile jest kolumn i najwyższa potęga jest od początku. Jeśli chodzi o kwestię interpretacji człowiek/aplikacja. Te wartości potęgują się, bo wtedy można precyzyjne rozpoznać, które wartości są dodane do zbioru poprzez agregację - po prostu wartość 1,2,3... zawsze będą oznaczały agregację/brak agregacji tych samych kolumn.

Np.

USE AdventureWorks

SELECT

p.Color, GROUPING\_ID(p.Color) AS 'Is color aggregated',

p.Size, GROUPING\_ID(p.Size) AS 'Is size aggregated',

p.Class, GROUPING\_ID(p.Class) AS 'Is class aggregated',

AVG(p.ListPrice) AS AveragePrice,

GROUPING\_ID(p.Color,p.Size,p.Class) AS "Aggregation mask"

FROM Production.Product AS p

GROUP BY GROUPING SETS ((p.Color), (p.Size), (P.Class))

**35. INNER JOIN**

**Złączenie tabel** → pisanie zapytań aby w wyniku tego zapytania przedstawić kolumny pochodzące z kilku różnych tabel. Należy skorzystać z faktu relacji pomiędzy bazami danych.

**Składnia ANSI-89** → sposób pierwszy do łączenia kolumn z różnych tabel. Rozwiązanie to oczywiście wymaga istnienia wspólnego klucza. To popularna składnia wykorzystywana w różnych językach bazodanowych. Zakłada ona wybór (SELECT) interesujących cię zaaliasowanych rekordów, FROM zaaliasowanie tabel WHERE klucz podstawowy jednej tabeli = klucz podstawowy drugiej tabeli. Jest to jednak rozwiązanie niedoskonałe gdyż klauzula WHERE określa nie tylko warunki filtrowania rekordów, ale też warunki jak mają zostać złączone tabele. Zatem problem polega na tym, że warunki łączące tabele są wymienione w klauzuli WHERE i trudno rozpoznać, który warunek odpowiada za połączenie tabel, a który za zwykłe filtrowanie danych.

Np.

USE AdventureWorks

SELECT

e.BusinessEntityID,

e.JobTitle,

p.FirstName,

p.LastName

FROM HumanResources.Employee AS e, Person.Person AS p

WHERE e.BusinessEntityID = p.BusinessEntityID

**Składnia ANSI-92** → składnia ulepszona wobec ANSI-89, korzysta ona z klauzuli JOIN i regułami klauzuli ON. Składnia ta określa reguły łączenia do tabeli a, tabeli b. Klauzula FROM odnosi się tutaj tylko do jednej tabeli, a reguły JOIN i ON określają która tabela ma na jakich warunkach być dodana do tej pierwszej.

 Np.

USE AdventureWorks

SELECT

e.BusinessEntityID,

e.JobTitle,

p.FirstName,

p.LastName

FROM HumanResources.Employee AS e

JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID

**Różnica między JOIN, a INNER JOIN w łączeniu tabel** → zwróć uwagę, że łącząc dwie tabele za pomocą JOIN i INNER JOIN mają w przykładzie takie same wyniki - oddają tą samą ilość rekordów. Pamiętaj przy tym, że jedna z tych tabel ma więcej rekordów niż druga, ale w tej pierwszej nie wszystkie rekordy mają klucz za pomocą którego można przypisać obie tabele. Zatem te rekordy są ignorowane - dzieje się to w przypadku JOIN i INNER JOIN. Nie ma to zatem znaczenia w tym kontekście z, którego rozwiązania skorzystasz.

 Np.

USE AdventureWorks

SELECT

e.BusinessEntityID,

e.JobTitle,

p.FirstName,

p.LastName

FROM HumanResources.Employee AS e

INNER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID

--JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID

**INNER JOIN i HAVING** → wyrażenie filtrujące HAVING znajduje się oczywiście na końcu zapytania.

Np.

USE AdventureWorks

SELECT

CONCAT(p.FirstName, '', p.LastName) AS 'Full Name',

COUNT(\*) AS 'Number of shifts'

FROM

HumanResources.EmployeeDepartmentHistory AS h

INNER JOIN Person.Person AS p ON h.BusinessEntityID = p.BusinessEntityID

GROUP BY CONCAT(p.FirstName, '', p.LastName)

HAVING COUNT(\*) > 1

**36. OUTER JOIN**

**LEFT/RIGHT OUTER JOIN** → zachowuje się inaczej jeśli nie potrafi znaleźć pasującego rekordu w drugiej tabeli (niż zachowałby się INNER JOIN). Chodzi o to, że jak podajesz tabele to INNER JOIN zwróci tylko dane tabel dla których dopasuje oba klucze. OUTER JOIN, jeśli tylko kolejność podawanych tabel będzie różna, to wynik będzie różny. Jeśli podasz większą tabelę, której rekordy pokrywają się z dodatkową, mniejszą jako pierwszą to przypisane zostaną rekordy z drugiej plus te, które nie mogły być przypisane. W takim wypadku dla nieistniejących rekordów tabeli drugiej pojawi się NULL. Różnica między LEFT i RIGHT: docelowo korzystaj z LEFT, które mówi, że tabelą wyjściową ma być w kodzie ta, która znalazła się po lewej stronie. Jeśli zastosujesz RIGHT to zamienisz ich kolejność co da inny wynik. W przypadku tej funkcji możesz pisać LEFT albo RIGHT JOIN bez OUTER i też zadziała.

Np.

USE AdventureWorks

SELECT

e.BusinessEntityID,

e.JobTitle,

p.FirstName,

p.LastName

FROM

Person.Person AS p

LEFT OUTER JOIN HumanResources.Employee AS e ON e.BusinessEntityID = p.BusinessEntityID

--Odwrotna kolejność tabel.

USE AdventureWorks

SELECT

e.BusinessEntityID,

e.JobTitle,

p.FirstName,

p.LastName

FROM

HumanResources.Employee AS e

LEFT OUTER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID

**Wykorzystywanie OUTER JOIN jako filtra** → jeśli masz dwie tabele i jedna z nich ma zdecydowanie mniej rekordów, możesz chcież wykorzystać funkcję OUTER JOIN jako filtr, a dla wartości NULL wstawić inną, określoną przez ciebie wartość za pomocą funckji ISNULL.

 Np.

USE AdventureWorks

SELECT

p.ProductID,

p.[Name],

ISNULL(pr.Comments, '---')

FROM

Production.Product AS p

LEFT JOIN Production.ProductReview AS pr

ON p.ProductID = pr.ProductID

**Dodawanie warunków do OUTER JOIN i wykorzystanie wielu OUTER JOIN** → jak widać na poniższym przykładzie można korzystać z więcej niż dwóch tabel naraz jeśli tylko połączy się je ze sobą. Tabela 1, 2 i 3: nawet jeśli tabela 1 nie jest bezpośrednio połączona z 3 to jeśli ich wspólnym mianownikiem jest 2 to można je połączyć. Ponadto jeśli chodzi o warunki WHERE to warto w ich miejsce wstawiać warunlo do klauzuli JOIN ON - chodzi o to, że WHERE kiedy nie dojdzie do skutku zwraca NULL.

USE Adventureworks

SELECT

p.[Name],

SUM(sod.OrderQty) AS 'SumAmount',

SUM(sod.OrderQty \* sod.UnitPrice) AS 'SumValue'

FROM Production.Product AS p

LEFT JOIN Sales.SalesOrderDetail AS sod ON sod.ProductID = p.ProductID

LEFT JOIN Sales.SalesOrderHeader AS soh ON soh.SalesOrderID = sod.SalesOrderID

AND soh.OrderDate BETWEEN '2014-05-01' AND '2014-05-31'

GROUP BY p.[Name]

**Łączenie dwóch tabel za pomocą więcej niż jednej pasującej pary kolumn oraz CASE** → na poniższym przykładzie widać, że chcę połączyć tabele żeby przy odpowiednim CASE zwracać, że określone wartości się pojawiają lub nie. Jest jeden haczyk. Tabela z której biorę informacje tylko o występowaniu nazw/skrótów dla wielkości/wagi produktu jest właśnie tak skonstruowana - ma wagę/rozmiar w jednej kolumnie, podczas gdy tabela produktów ma osobne kolumny dla wagi/rozmiaru. W związku z tym muszę zaaliasować tabelę z dwoma osobnymi kolumnami (Product) dwa razy i podać ją dwa razy w klauzuli OUTER JOIN. Zwróć jednocześnie uwagę, że aliasowanie nie musi oznaczać jednej literki, może ich być więcej. Zwróć też uwagę, że wykorzystałeś prostą instrukcję CASE żeby dodać jedną kolumnę sprawdzającą pojawianie się lub nie tych samych symboli w kolumnie tabeli z jednostkami miary oraz tabeli z dwoma osobnymi kolumnami waga/wielkość.

Np.

USE AdventureWorks

SELECT

m.[Name],

m.UnitMeasureCode,

CASE

WHEN pSize.SizeUnitMeasureCode IS NOT NULL THEN 'Is used as size'

WHEN pWeight.WeightUnitMeasureCode IS NOT NULL THEN 'Is used as weight'

END AS 'Used as'

FROM Production.UnitMeasure AS m

LEFT JOIN Production.Product AS pSize ON m.UnitMeasureCode = pSize.SizeUnitMeasureCode

LEFT JOIN Production.Product AS pWeight ON m.UnitMeasureCode = pWeight.WeightUnitMeasureCode

WHERE pSize.SizeUnitMeasureCode IS NULL AND pWeight.WeightUnitMeasureCode IS NULL

**37. ZŁĄCZENIA WIELU TABEL**

**Złączenia wielu tabel** → możesz łączyć więcej niż dwie tabele naraz. Niezbędne do łączenia tabeli jest to, żeby istniała między nimi ciągła relacja. Żeby zobaczyć relacje stwórz diagram i dla wszystkich tabeli połączonych ciągiem kluczy możesz połączyć kolumny. Robisz to następująco: wstawiasz wiele JOIN-ów na zakładkę. A = B, B = C, C = D, D=E, itd. Wstawiaj JOIN-y po kolei, czyli od tabeli, która jest nadrzędna tematycznie (np. Person.Person jest taką tabelą).

Np.

USE AdventureWorks

SELECT

P.BusinessEntityID,

P.FirstName,

P.LastName,

BE.BusinessEntityID,

BEA.BusinessEntityID,

A.City,

A.PostalCode,

A.AddressLine1,

A.AddressLine2

FROM

Person.Person AS P

JOIN Person.BusinessEntity AS BE ON P.BusinessEntityID = BE.BusinessEntityID

JOIN Person.BusinessEntityAddress AS BEA ON BE.BusinessEntityID = BEA.BusinessEntityID

JOIN Person.Address AS A ON BEA.AddressID = A.AddressID

**Usuwanie nie interesujących informacji** → przy łączeniu tabel nie musisz wskazywać wszystkich kolumn z kluczami, możesz je eliminować. Nie musisz koniecznie wyświetlać zawartości tabel, które są łączone, a które są niezbędne do działania danej relacji.

Np.

USE AdventureWorks

SELECT

P.FirstName,

P.LastName,

A.City,

A.PostalCode,

A.AddressLine1,

A.AddressLine2

FROM

Person.Person AS P

JOIN Person.BusinessEntity AS BE ON P.BusinessEntityID = BE.BusinessEntityID

JOIN Person.BusinessEntityAddress AS BEA ON BE.BusinessEntityID = BEA.BusinessEntityID

JOIN Person.Address AS A ON BEA.AddressID = A.AddressID

**Łączenie kolumn o różnych nazwach** → czyli jak na przykładzie P.BusinessEntityID = SOH.SalesPersonID. Zwróć uwagę, że chodzi o to, że wartości kolumny 1 mają być równe wartościom kolumny 2. Dwie kolumny nie muszą mieć takich samych nazw, żeby mieć wspólny klucz.

USE AdventureWorks

SELECT

P.FirstName,

P.LastName,

PC.[Name],

SUM(SOD.OrderQty \* SOD.UnitPrice) AS 'TotalValue'

FROM

Sales.SalesOrderHeader AS SOH

JOIN Person.Person AS P ON P.BusinessEntityID = SOH.SalesPersonID

JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID

JOIN Production.Product AS PRD ON PRD.ProductID = SOD.ProductID

JOIN Production.ProductSubcategory AS PS ON PS.ProductSubcategoryID = PRD.ProductSubcategoryID

JOIN Production.ProductCategory AS PC ON PC.ProductCategoryID = PS.ProductCategoryID

WHERE SOH.OrderDate = '2011-05-31'

GROUP BY P.FirstName, P.LastName, PC.[Name]

–Poniżej osobna prezentacja dwóch kolumn o różnych nazwach, ale wspólnym kluczu.

USE AdventureWorks

SELECT

P.BusinessEntityID,

SOH.SalesPersonID

FROM

Sales.SalesOrderHeader AS SOH

JOIN Person.Person AS P ON P.BusinessEntityID = SOH.SalesPersonID

**LEFT JOIN i GROUP BY** → jak widać na poniższym przykładzie w ramach łączenia tabel możesz używać LEFT JOIN w celu ujawniania NULL-i i GROUP BY w celu grupowania rekordów

USE AdventureWorks

SELECT

PP.[Name] AS 'Product name',

MAX(SSOH.OrderDate) AS 'Last ordered'

FROM

Production.Product AS PP

LEFT JOIN Sales.SalesOrderDetail AS SSOD ON PP.ProductID = SSOD.ProductID

LEFT JOIN Sales.SalesOrderHeader AS SSOH ON SSOD.SalesOrderID = SSOH.SalesOrderID

GROUP BY PP.[Name]

ORDER BY 'Last ordered' DESC

**38. CROSS JOIN**

**CROSS JOIN** → polecenie łączące rekord z kolumny z każdym możliwym dla niego rekordem z drugiej tabeli/trzeciej tabeli. Polecenie nie używa klauzuli ON i wyznacza które kolumny, której tabeli wymieniają wszystkie interesujące rekordy, które mają być skrzyżowane w tymi z pozostałych tabel.

Np.

USE AdventureWorks

SELECT

sp.BusinessEntityID,

p.FirstName,

p.LastName,

st.TerritoryID,

st.[Name]

FROM Sales.SalesPerson AS sp

CROSS JOIN Sales.SalesTerritory AS st

JOIN Person.Person AS p ON p.BusinessEntityID = sp.BusinessEntityID

**Znak różności** → czyli <>. Jeśli wstawisz ten znak między kolumnami w klauzuli WHERE to oznacza to, że wyszukiwane i wyświetlane są rekordy różne niż coś i coś, czyli nie mogą być takie same.

Np.

USE AdventureWorks

SELECT

DISTINCT s1.City, s2.City

FROM

Sales.vStoreWithAddresses AS s1

CROSS JOIN Sales.vStoreWithAddresses AS s2

WHERE

s1.CountryRegionName = 'Germany' AND s2.CountryRegionName = 'Germany'

AND s1.City <> s2.City

ORDER BY s1.City, s2.City

**CROSS JOIN i wyszukiwanie wszystkich możliwych par unikalnych rekordów** → możesz użyć CROSS JOIN jeśli chcesz zobaczyć wszystkie możliwe pary unikalnych rekordów. Użyj do tego funkcji DISTINCT, która wyciąga tylko unikalne rekordy. Na przykładzie poniżej zaaliasowałeś dwa razy tą samą tabelę, bo potrzebowałeś połączyć ze sobą wszystkie możliwe połączenia między miastami. Dodaj również znak różności, żeby nie połączyć ze sobą dwóch tych samych rekordów ALBO możemy postawić znak większe. To drugie rozwiązanie sprawi, że nie tylko rekordy się nie zdublują, ale również unikalne połączenia się nie pojawią, bo w porządku alfabetycznym Jedna nazwa zawsze będzie większa od drugiej. Inaczej: rozwiązanie <> nie dopuści Berlin-Berlin, ale dopuści Berlin-Bonn i Bonn-Berlin, a rozwiązanie > nie dopuści ani Berlin-Berlin i dopuści tylko Berlin-Bonn albo Bonn-Berlin w zależności co jest wyżej w alfabecie.

Np.

USE AdventureWorks

SELECT

DISTINCT s1.City, s2.City

FROM

Sales.vStoreWithAddresses AS s1

CROSS JOIN Sales.vStoreWithAddresses AS s2

WHERE

s1.CountryRegionName = 'Germany' AND s2.CountryRegionName = 'Germany'

AND s1.City > s2.City

ORDER BY s1.City, s2.City

**DISTINCT i sposób zapisu** → zwróć uwagę, że w przykładzie poniżej funkcja DISTINCT nie ma argumentów w nawiasie, a one następują jedna po drugiej. Jest to sposób zapisu tej funkcji.

USE AdventureWorks

SELECT

DISTINCT pp\_1.Color, pp\_2.Color

FROM

Production.Product AS pp\_1

CROSS JOIN Production.Product AS pp\_2

WHERE pp\_1.Color > pp\_2.Color

**CROSS JOIN i JOIN** → czasami istnieje potrzeba żeby w jednym zapytaniu skorzystać jednocześnie z CROSS JOIN i JOIN. Jeśli chcesz w krzyżowy sposób połączyć dane z jednej tabeli, np. dobrać we wszystkie możliwe pary pracowników, ale jednocześnie musisz z innej tabeli wyciągać szczegóły rekordów (np. imię i nazwisko), możesz skorzystać z obu instrukcji. Wzór kolejności: JOIN - CROSS JOIN - JOIN

Np.

USE AdventureWorks

SELECT

p1.LastName, p1.FirstName

,p2.LastName,p2.FirstName

FROM Sales.SalesPerson s1

JOIN Person.Person p1 ON p1.BusinessEntityID = s1.BusinessEntityID

CROSS JOIN Sales.SalesPerson s2

JOIN Person.Person p2 ON p2.BusinessEntityID = s2.BusinessEntityID

WHERE s1.BusinessEntityID < s2.BusinessEntityID

**39. FULL JOIN**

**FULL JOIN** → ma on za zadanie dodać wyniki LEFT JOIN i RIGHT JOIN. Pisownia: FULL OUTER JOIN lub FULL JOIN. Zwróć uwagę na różnice trzech zapisów na poniższym przykładzie: JOIN łączy wszystkie rekordy, których połączenie nie zwróci NULL; LEFT JOIN pokaże to co JOIN i rekordy znajdujące się w "lewej tabeli" dla których nie było pasujących rekordów w drugiej, "prawej" tabeli; ostatecznie FULL JOIN pokazuje to wszystko i również rekordy z "prawej" czyli drugiej tabeli, dla których nie ma pasujących rekordów z tabeli pierwszej. Czyli jak na przykładzie: jednostki miary dla których nie było produktów.

USE AdventureWorks

SELECT

p.[Name],

p.SizeUnitMeasureCode,

um.UnitMeasureCode,

um.[Name]

FROM

Production.Product AS p

--JOIN Production.UnitMeasure AS um ON p.SizeUnitMeasureCode = um.UnitMeasureCode

--LEFT JOIN Production.UnitMeasure AS um ON p.SizeUnitMeasureCode = um.UnitMeasureCode

FULL JOIN Production.UnitMeasure AS um ON p.SizeUnitMeasureCode = um.UnitMeasureCode

ORDER BY p.SizeUnitMeasureCode DESC

**LEFT JOIN i która tabela jest lewa, a która prawa** → LEFT JOIN zwraca wszystkie rekordy z lewej tabeli (tabela\_1) i pasujące rekordy z prawej tabeli (tabela\_2). Wynikiem jest 0 rekordów z prawej strony, jeśli nie ma dopasowania. Tabela\_1 to tabela znajdująca się za FROM, a tabela\_2 to co za LEFT JOIN.

Np.

USE AdventureWorks

SELECT kolumna\_nazwa

FROM tabela\_1

LEFT JOIN tabela\_2

ON tabela\_1.kolumna\_nazwa = tabela\_2.kolumna\_nazwa

**BEGIN TRANSACTION, ROLLBACK oraz COMMIT** → Jeśli dodasz BEGIN TRANSACTION (lub BEGIN TRAN) przed instrukcją, automatycznie uczyni ona transakcję jawną i zablokuje tabelę do czasu zatwierdzenia lub wycofania transakcji. Na przykład, gdy wydaję instrukcję DELETE lub UPDATE, mogę użyć BEGIN TRAN, aby upewnić się, że moja instrukcja jest poprawna i otrzymuję poprawną liczbę zwracanych wyników. Ponieważ określiłem BEGIN TRAN, transakcja oczekuje teraz na ROLLBACK lub COMMIT. Podczas gdy transakcja oczekuje, utworzyła blokadę , a wszystkie inne procesy próbujące uzyskać dostęp do HumanResources.Employee są teraz blokowane. Zachowaj ostrożność, używając BEGIN TRAN i upewnij się, że natychmiast wydałeś polecenie ROLLBACK lub COMMIT.

Np.

UPDATE HumanResources.Employee

SET JobTitle = ‘DBA’

WHERE LoginID IN (SELECT LoginID FROM HumanResources.Employee)

--Inny przykład

USE AdventureWorks

BEGIN TRAN

ALTER TABLE Person.Person NOCHECK CONSTRAINT ALL

ALTER TABLE Sales.SalesPerson NOCHECK CONSTRAINT ALL

ALTER TABLE HumanResources.EmployeeDepartmentHistory NOCHECK CONSTRAINT ALL

ALTER TABLE HumanResources.EmployeePayHistory NOCHECK CONSTRAINT ALL

ALTER TABLE HumanResources.JobCandidate NOCHECK CONSTRAINT ALL

ALTER TABLE HumanResources.Employee NOCHECK CONSTRAINT ALL

UPDATE HumanResources.Employee set BusinessEntityID=1074

WHERE BusinessEntityID = 274

--ROLLBACK na końcu

ROLLBACK

**40. Podzapytania skalarne**

**Podzapytania skalarne** → SELECT wybiera określone dane, a JOIN łączy tabele. Podzapytanie to zapytanie, które znajduje się w innym, szerszym zapytaniu. Wzór: piszesz je po przecinku w nawiasie, cały jego zapis. W podzapytanaich można oczywiście dokonywać obliczeń. Zwróć uwagę, że jeśli dwa razy korzystasz w różnych podzapytaniach z tych samych obliczeń to nie możesz w takim wypadku skorzystać z aliasów. Jednocześnie zauważ, że można korzystać z JOIN, a w ORDER BY został wykorzystany alias z jednego z podzapytań - działa on ponieważ w zapytaniu SELECT najpierw dochodzi do aliasowania, a dopiero potem do grupowania. W przypadku wykorzystania klauzuli WHERE również trzeba napisać całe zapytanie.

Np.

–Pierwsza część zapytania osobno bez podzapytania

USE AdventureWorks

SELECT

sp.BusinessEntityID,

sp.Bonus

FROM

Sales.SalesPerson AS sp

--Druga część zapytania osobno bez podzapytania

USE AdventureWorks

SELECT

AVG(sp.Bonus)

FROM Sales.SalesPerson AS sp

--Zapytanie z podzapytaniami

USE AdventureWorks

SELECT

sp.BusinessEntityID,

p.LastName,

p.FirstName,

sp.Bonus,

(SELECT AVG(sp.Bonus) FROM Sales.SalesPerson AS sp) AS 'Average bouns',

sp.Bonus - (SELECT AVG(sp.Bonus) FROM Sales.SalesPerson AS sp) AS 'Difference in bonus',

100 \* (sp.Bonus/(SELECT SUM(sp.Bonus) FROM Sales.SalesPerson AS sp)) AS 'Percent of total'

FROM

Sales.SalesPerson AS sp

JOIN Person.Person AS p ON p.BusinessEntityID = sp.BusinessEntityID

WHERE sp.Bonus > (SELECT AVG(sp.Bonus) FROM Sales.SalesPerson AS sp)

ORDER BY 'Percent Of Total' DESC

**Podzapytania zamiast tworzenia zmiennych** → czasami zamiast tworzyć zmienną, którą odkryłem pisząc odrębne zapytanie mogę skorzystać z podzapytania, które właśnie oddaje prawidłową odpowiedź. Tak jak w przykładzie poniżej, zapytanie wewnętrzne oddaje 5000 i wiedząc to za znakiem równości mogłem po prostu wstawić 5000, ale może nie chcę tworzyć zmiennych, zatem wykorzystam podzapytanie.

Np.

USE AdventureWorks

SELECT

sp.BusinessEntityID,

sp.Bonus

FROM Sales.SalesPerson AS sp

WHERE

sp.Bonus = (SELECT Bonus FROM Sales.SalesPerson WHERE BusinessEntityID = 280)

**Działania na podzapytaniach** → zwróć uwagę, że żeby policzyć coś za pomocą podzapytania, gdzie korzystasz z podzapytania i danych "zwykłych" z tabeli, na przykład odejmujesz je od siebie to te zwykłe dane nie znajdują się w podzapytaniu - jakby się znalazły to SQL prawdopodobnie poprosiłby o utworzenie GROUP BY.

USE AdventureWorks

SELECT

hre.LoginID,

hre.SickLeaveHours,

(SELECT AVG(hre.SickLeaveHours) FROM HumanResources.Employee AS hre) AS AvgSickLeaveHours,

hre.SickLeaveHours - (SELECT AVG(hre.SickLeaveHours) FROM HumanResources.Employee AS hre) AS AvgSickLeaveDiff

FROM

HumanResources.Employee AS hre

WHERE hre.SickLeaveHours > (SELECT AVG(hre.SickLeaveHours) FROM HumanResources.Employee AS hre)

ORDER BY AvgSickLeaveDiff DESC

**41. Podzapytania zwracające wiele wartości**

**Podzapytania zwracające wiele wartości i składnia IN** → jeśli chcę za pomocą podzapytania filtrować dane, a zatem wyświetlić więcej niż jedną informację naraz, tylko zakres informacji, zakres warunków, które jeśli spełnione to zostaną wyświetlone, mogę w klauzuli WHERE to zrobić korzystając z podzapytania i instrukcji IN. Zwróć uwagę, że w zapytaniu całym znajdują się odwołania do dwóch tabel. tylko w podzapytaniu znajduje się odwołanie do innej tabeli, ale tak naprawdę wynikiem, który jest wykorzystany w klauzuli WHERE "nadzapytania" są liczby - ProductID, które są zbieżne dla obu tabel.

Np.

USE AdventureWorks

SELECT

\*

FROM

Sales.SalesOrderDetail AS sod

WHERE

sod.ProductID IN (SELECT ProductID FROM Production.Product WHERE Color = 'Multi')

ORDER BY sod.ProductID

**Podzapytania i alternatywa czyli JOIN** → czasami to samo można osiągnąć różnymi metodami. Poniższe przykłady pokazują jak dwa zapytania (jedno z podzapytaniem, a drugie z JOIN) ostatecznie zwracają taki sam wynik.

Np.

--Podzapytanie

USE AdventureWorks

SELECT

\*

FROM

Sales.SalesOrderDetail AS sod

WHERE

sod.ProductID IN (SELECT ProductID FROM Production.Product WHERE Color = 'Multi')

ORDER BY sod.ProductID

--JOIN

USE AdventureWorks

SELECT

\*

FROM

Sales.SalesOrderDetail AS sod

JOIN Production.Product AS pp ON sod.ProductID = pp.ProductID

WHERE

pp.Color = 'Multi'

ORDER BY sod.ProductID

**Różnica między JOIN, a podzapytaniami - Display Estimated Execution Plan** → ogólnie przyjmuje się, że JOIN-y są lepsze. Możesz ponadto sprawdzić co jest mniej skomplikowane i kosztowne do obliczenia dla SQL korzystając z Display Estimated Execution Plan (skrót Crtl+L). Poniższy przykład pokaże, że są takie same.

Np.

SELECT

\*

FROM

Sales.SalesOrderDetail AS sod

WHERE

sod.ProductID IN (SELECT ProductID FROM Production.Product WHERE Color = 'Multi')

ORDER BY sod.ProductID

SELECT

\*

FROM

Sales.SalesOrderDetail AS sod

JOIN Production.Product AS pp ON sod.ProductID = pp.ProductID

WHERE

pp.Color = 'Multi'

ORDER BY sod.ProductID

**42. Podzapytania wykorzystywane jako wirtualne tabele**

**Podzapytania wykorzystywane jako wirtualne tabele** → jest to pewna alternatywa na JOIN. Mogę pisać całe podzapytanie SELECT, zaaliasować je jako coś, jeden element, który zostanie w "nadzapytaniu" SELECT dodany jako właśnie jeden element - a zatem ten jeden element może dodać wiele kolumn z różnych tabel, które zostały określone w podzapytaniu, czyli tzw. "wirtualnej tabeli". I można osiągnąć to samo za pomocą dodawania kolejnych elementów przez JOIN, ale jeśli w przyszłości okazałoby się, że struktury tabel ulegną zmianie albo, że pojawi się błąd, to modyfikacja kodu aplikacji będzie stosunkowo prosta, bo wystarczy wymienić wewnętrzne zapytanie zwracające adres osób. Ponadto zwróć uwagę, że takie podzapytanie możesz wykorzystywać w wielu zapytaniach, jeśli tylko klucze są kompatybilne. W przykładzie dwa sposoby wykorzystania tego samego podzapytania. Struktura takiego zapytania do tabela wyjściowa + podzapytanie (tabela wirtualna). Pamiętaj też, że w SELECT "nadrzędnym" korzystasz z gwiazdki odwołując się do podzapytania, bo bierzesz z niego raczej wszystko. Zwróć uwagę też, że w klauzuli JOIN, tej zewnętrznej odwołujesz się nie do aliasów tabel podzapytania, ale do aliasu nadanego podzapytaniu + kolumn wyłonionych w podzapytaniu.

Np.

USE AdventureWorks

SELECT

emp.BusinessEntityID,

emp.HireDate,

emp.LoginID,

Addr.\*

--Powyższe oznacza, że biorę wszystko, że aliasuję Addr i biorę z tego wszystko

FROM HumanResources.Employee AS emp

JOIN

(SELECT

p.BusinessEntityID,

p.LastName,

p.FirstName,

pat.[Name],

a.PostalCode,

a.City,

a.AddressLine1,

a.AddressLine2

FROM

Person.Person AS p

JOIN Person.BusinessEntityAddress AS bea ON bea.BusinessEntityID = p.BusinessEntityID

JOIN Person.Address AS a ON a.AddressID = bea.AddressID

JOIN Person.AddressType AS pat ON bea.AddressTypeID = pat.AddressTypeID

WHERE pat.[Name] = 'Home')

--Całe powyższe SELECT podzapytania zostało zaaliasowane jako Addr w kolejnej linijce kodu

AS Addr ON Addr.BusinessEntityID = emp.BusinessEntityID

--Poniżej inne zapytanie, ale wykorzystujące to samo podzapytanie

USE AdventureWorks

SELECT

Addr.\*

FROM HumanResources.JobCandidate AS jc

JOIN

(SELECT

p.BusinessEntityID,

p.LastName,

p.FirstName,

pat.[Name],

a.PostalCode,

a.City,

a.AddressLine1,

a.AddressLine2

FROM

Person.Person AS p

JOIN Person.BusinessEntityAddress AS bea ON bea.BusinessEntityID = p.BusinessEntityID

JOIN Person.Address AS a ON a.AddressID = bea.AddressID

JOIN Person.AddressType AS pat ON bea.AddressTypeID = pat.AddressTypeID

WHERE pat.[Name] = 'Home')

AS Addr ON Addr.BusinessEntityID = jc.BusinessEntityID

**Zastosowanie podzapytań jako wirtualnych tabel w praktyce** → takie wirtualne tabele przydają się gdy trzeba wykonać daną operację na wielu różnych tabelach albo gdy wiesz, że będą ci te dane potrzebne w przyszłości to możesz taką wirtualną tabelę przygotować.

Np.

USE AdventureWorks

SELECT

pp.ProductID,

pp.[Name] AS 'ProductName',

ppc.[Name] AS 'ProductCategoryName',

pps.[Name] AS 'ProductSubcategoryName'

FROM

Production.Product AS pp

JOIN Production.ProductSubcategory AS pps ON pp.ProductSubcategoryID = pps.ProductSubcategoryID

JOIN Production.ProductCategory AS ppc ON pps.ProductCategoryID = ppc.ProductCategoryID

**43. Podzapytania skorelowane**

**Podzapytania nieskorelowane** → najpierw co to jest zapytanie nieskorelowane. To podzapytanie niezależne od zapytania zewnętrznego, możesz je teoretycznie wybrać i uruchomić niezależnie.

Np.

USE AdventureWorks

SELECT \*

FROM Production.Product AS pp

WHERE pp.ProductID IN (SELECT ssd.ProductID FROM Sales.SalesOrderDetail AS ssd WHERE ssd.SalesOrderID = 43661)

Podzapytania skorelowane --> podzapytanie skorelowane to takie, które jest zależne od zewnętrznego, czyli wewnętrzne/podzapytanie jest zależne od zewnętrznego/nadrzędnego, chociażby dlatego, że w podzapytaniu wykorzystywany jest alias utworzony w zapytaniu zewnętrznym.

Np.

USE AdventureWorks

SELECT p1.ProductID,

p1.[Name],

p1.ListPrice,

p1.ProductSubcategoryID,

(SELECT AVG(p2.ListPrice) AS 'AvgInSubcategory'

FROM Production.Product AS p2

WHERE p2.ProductSubcategoryID = p1.ProductSubcategoryID)

FROM Production.Product AS p1

WHERE (SELECT AVG(p2.ListPrice) AS 'AvgInSubcategory'

FROM Production.Product AS p2

WHERE p2.ProductSubcategoryID = p1.ProductSubcategoryID)

<= p1.ListPrice

**Podzapytania skorelowane, a różne tabele** → pisząc podzapytanie skorelowane nie trzeba koniecznie odwoływać się do tych samych tabel, mogą one być różne. Poniżej przykład wykorzystujący sprzedaże danych pracowników i danych terytoriów.

USE AdventureWorks

SELECT

sp.BusinessEntityID,

sp.SalesLastYear

FROM Sales.SalesPerson AS sp

WHERE

(SELECT COUNT(\*) FROM Sales.SalesTerritory AS st

WHERE st.SalesLastYear < sp.SalesLastYear) > 0

**WHERE w podzapytaniu skorelowanym** → zwróć uwagę, że w poniższym przykładzie znajduje się klauzula WHERE w podzapytaniu skorelowanym, która wyciąga interesujące cię produkty o ostatniej (maksymalnej) dacie sprzedaży. W ten sposób, za pomocą tego WHERE i znaku równości między tą samą tabelą (zaaliasowaną podwójnie, raz wewnątrz, raz zewnątrz) w zapytaniu zewnętrznym i wewnętrznym upewniasz się, że zapytanie SELECT o najwyższą datę nie odnosić się będzie do najwyższej daty całej tabeli, ale konkretnego produktu (po ProductID).

USE AdventureWorks

SELECT

DISTINCT sod1.ProductID, soh1.OrderDate

FROM Sales.SalesOrderDetail AS sod1

JOIN Sales.SalesOrderHeader soh1 ON soh1.SalesOrderID = soh1.SalesOrderID

WHERE

soh1.OrderDate = (

SELECT MAX(soh2.OrderDate) FROM Sales.SalesOrderHeader soh2

JOIN Sales.SalesOrderDetail sod2 ON sod2.SalesOrderID = soh2.SalesOrderID

WHERE sod2.ProductID = sod1.ProductID

)

**44. Podzapytania EXISTS, ALL, SOME, ANY**

**EXISTS** → klauzula sprawdzająca czy istnieje co najmniej jeden rekord spełniający określony warunek. Możesz np. sprawdzać czy wśród wszystkich rekordów (COUNT(\*)) danej tabeli znajduje się choć jeden znajdujący swój odpowiednik (ProductID = ProductID) w innej tabeli (klauzula WHERE + EXISTS). Wynikiem EXISTS będzie ile rekordów spełnia określony warunek. Warunek jest w podzapytaniu.

Np.

USE AdventureWorks

SELECT COUNT(\*)

FROM Production.Product p

WHERE EXISTS (SELECT \* FROM Sales.SalesOrderDetail sod WHERE sod.ProductID = p.ProductID)

**NOT EXISTS** → klauzula sprawdzająca czy NIE ISTNIEJE co najmniej jeden rekord spełniający określony warunek. Możesz np. sprawdzać czy wśród wszystkich rekordów (COUNT(\*)) danej tabeli znajduje się choć jeden NIE ZNAJDUJĄCY swój odpowiednik (ProductID = ProductID) w innej tabeli (klauzula WHERE + NOT EXISTS). Wynikiem NOT EXISTS będzie ile rekordów NIE SPEŁNIA określonego warunku. Warunek jest w podzapytaniu.

Np.

USE AdventureWorks

SELECT COUNT(\*)

FROM Production.Product p

WHERE NOT EXISTS (SELECT \* FROM Sales.SalesOrderDetail sod WHERE sod.ProductID = p.ProductID)

**EXISTS/NOT EXISTS, a wyświetlanie całych tabel** → jeśli chcę widzieć nie samą liczbę rekordów (COUNT), a całe tabele informacji o rekordach spełniających/niespełniających warunków EXISTS, wystarczy w klauzul WHERE dodać właśnie EXISTS lub NOT EXISTS.

Np.

USE AdventureWorks

SELECT \*

FROM Production.Product p

WHERE EXISTS (SELECT \* FROM Sales.SpecialOfferProduct sop WHERE sop.ProductID = p.ProductID)

--WHERE NOT EXISTS (SELECT \* FROM Sales.SpecialOfferProduct sop WHERE sop.ProductID = p.ProductID)

**ALL** → klauzula pozwalająca na wyodrębnienie w klauzuli WHERE rekordów spełniających warunek zwracający wiele wyników - a właściwie wszystkich wyników spełniający warunek podzapytania. W zwykłym zapytaniu WHERE SQL zwraca błąd jeśli spróbujesz wykorzystać znak równości/większe/mniejsze/etc. wobec więcej niż jednej wartości, ale z klauzulą ALL jest to możliwe.

Np.

USE AdventureWorks

SELECT \*

FROM Production.Product p1

WHERE p1.DaysToManufacture > ALL (SELECT DaysToManufacture FROM Production.Product WHERE Color IS NULL)

--WHERE p1.DaysToManufacture <= ALL (SELECT DaysToManufacture FROM Production.Product WHERE Color IS NULL)

**SOME** → działa w przeciwną stronę jak ALL. Jest równoważna ANY. Wyświetla jakiekolwiek rekordy spełniające warunek podany w podzapytaniu SELECT. Wystarczy zatem, że wynikiem podzapytania będzie choć jeden rekord, który spełni warunek dla tysiąca innych, to ten tysiąc zostanie wyświetlony.

Np.

USE AdventureWorks

SELECT \*

FROM Production.Product p1

WHERE p1.DaysToManufacture <= SOME (SELECT DaysToManufacture FROM Production.Product WHERE Color IS NULL)

**ANY** → działa w przeciwną stronę jak ALL. Jest równoważna SOME. Wyświetla jakiekolwiek rekordy spełniające warunek podany w podzapytaniu SELECT. Wystarczy zatem, że wynikiem podzapytania będzie choć jeden rekord, który spełni warunek dla tysiąca innych, to ten tysiąc zostanie wyświetlony.

Np.

USE AdventureWorks

SELECT \*

FROM Production.Product p1

WHERE p1.DaysToManufacture <= ANY (SELECT DaysToManufacture FROM Production.Product WHERE Color IS NULL)

**45. Zastępowanie podzapytań przez JOIN**

**Zastępowanie podzapytań przez JOIN** → czasami wykorzystywanie zapytania JOIN jest bardziej optymalne dla SQL Server niż tworzenie podzapytań, wymaga mniej pracy, wykorzystania mniej pamięci. Całkiem spora ilość zapytań wykorzystujących podzapytania może być przepisana do postaci wykorzystującej polecenie JOIN. Ludziom częściej łatwiej będzie zrozumieć zapytanie, które zostanie zapisane z wykorzystaniem podzapytań aniżeli bazujące tylko i wyłącznie na klauzuli JOIN. Dodatkowo, jeśli zapytanie nie jest zbyt skomplikowane, to optymalizator zapytań potrafi je wykonać tak samo, jakby było zapisane poprzez join, innymi słowy w pewnych sytuacjach nieważne, czy zapytanie jest zapisane jako join czy jako podzapytanie. Zazwyczaj zapisanie złożonego zapytania z wieloma podzapytaniami do postaci JOIN poprawia wydajność wykonania zapytania.

Np.

–Wykorzystanie podzapytań

USE AdventureWorks

SELECT

DISTINCT pp.ProductID

,pp.Name

FROM Production.Product pp

WHERE pp.ProductID IN

(SELECT

DISTINCT sod.ProductID

FROM Sales.SalesOrderDetail sod

JOIN Sales.SalesOrderHeader soh ON soh.SalesOrderID = sod.SalesOrderID

WHERE soh.OrderDate BETWEEN '2013-07-01' AND '2013-07-31')

--Wykorzystanie JOIN

SELECT

DISTINCT p.ProductID, p.Name

FROM Production.Product p

JOIN Sales.SalesOrderDetail sod ON sod.ProductID = p.ProductID

JOIN Sales.SalesOrderHeader soh ON soh.SalesOrderID = sod.SalesOrderID

WHERE

soh.OrderDate BETWEEN '2013-07-01' AND '2013-07-31'

--Sytuacja odwrotna (NOT IN): Wykorzystanie podzapytań

USE AdventureWorks

SELECT

DISTINCT pp.ProductID

,pp.Name

FROM Production.Product pp

WHERE pp.ProductID NOT IN

(SELECT

DISTINCT sod.ProductID

FROM Sales.SalesOrderDetail sod

JOIN Sales.SalesOrderHeader soh ON soh.SalesOrderID = sod.SalesOrderID

WHERE soh.OrderDate BETWEEN '2013-07-01' AND '2013-07-31')

--Sytuacja odwrotna (LEFT OUTER JOIN): Wykorzystanie JOIN

SELECT \* FROM Production.Product p

LEFT OUTER JOIN

(

SELECT

DISTINCT sod.ProductID

FROM Sales.SalesOrderDetail sod

JOIN Sales.SalesOrderHeader soh ON soh.SalesOrderID = sod.SalesOrderID

WHERE

soh.OrderDate BETWEEN '2013-07-01' AND '2013-07-31'

) sold ON p.ProductID = sold.ProductID

WHERE sold.ProductID IS NULL

**46. Operator UNION**

**Operator UNION** → operator ten pozwala na połączenie wyników dwóch zapytań SELECT, tzn. ich wyniki nie będą widoczne obok siebie w osobnych kolumnach, ale w jednej kolumnie wymieszane. UNION znajdzie się między dwoma zapytaniami SELECT (linię przed drugim zapytaniem SELECT). Zwróć uwagę, że nazwy kolumn będą zgodne z nazwami pierwszego zapytania SELECT. Zauważ też, że liczba kolumn, których wartości są łączone musi być taka sama. Ponadto kolumny łączone są na zasadzie "SELECT Kolumna\_1, Kolumna\_2 (...) UNION SELECT Kolumna\_3, Kolumna\_4" gdzie kolumny 1 i 3 zostaną połączone oraz 2 i 4. Oznacza to, że te kolumny muszą mieć te same typy zmiennych.

Np.

USE AdventureWorks

SELECT

c.CustomerID 'Cus\_test', c.AccountNumber 'Acc\_1\_test'

FROM Sales.Customer c

UNION

SELECT

v.BusinessEntityID 'Bus\_test', v.AccountNumber 'Acc\_2\_test'

FROM Purchasing.Vendor v

**Pusta kolumna, a operator UNION** → dodając kolejne kolumny do UNION należy pamiętać, że liczba łączonych kolumn musi być równa po obu stronach UNION. Jeśli to niemożliwe bo np. jedna tabela ma dostęp do imion i nazwisk, a druga tylko imion, należy w miejsce nazwiska dla drugiej tabeli wstawić w SELECT po prostu ''.

Np.

USE AdventureWorks

SELECT

c.CustomerID 'Cus\_test', c.AccountNumber 'Acc\_1\_test', p.LastName, p.FirstName

FROM Sales.Customer c

LEFT JOIN Person.Person p ON c.CustomerID = p.BusinessEntityID

UNION

SELECT

v.BusinessEntityID 'Bus\_test', v.AccountNumber 'Acc\_2\_test', v.Name, ''

FROM Purchasing.Vendor v

**Porządkowanie kolejności wartości zwróconych przez UNION** → można to zrobić za pomocą DESC/ASC. Musi jednak pojawić się tylko jeden raz i tylko na końcu wszystkich zapytań z UNION.

Np.

USE AdventureWorks

SELECT v.BusinessEntityID, v.AccountNumber, v.Name AS 'Name or LastName', '' AS 'FirstName'

FROM Purchasing.Vendor v

UNION

SELECT c.CustomerID, c.AccountNumber, p.LastName, p.FirstName

FROM Sales.Customer c

LEFT JOIN Person.Person p ON c.CustomerID = p.BusinessEntityID

ORDER BY 'Name or LastName' DESC

**Łączenie więcej niż 2 (od 3) podzapytań** → jest to możliwe, jedynie między zapytaniami SELECT trzeba umieścić UNION.

Np.

USE AdventureWorks

SELECT

c.CustomerID, c.AccountNumber, a.City, a.PostalCode, a.AddressLine1

FROM Sales.Customer c

JOIN Person.BusinessEntityAddress ba ON ba.BusinessEntityID = c.CustomerID

JOIN Person.Address a ON a.AddressID = ba.AddressID

UNION

SELECT

v.BusinessEntityID, v.AccountNumber, a.City, a.PostalCode, a.AddressLine1

FROM Purchasing.Vendor v

JOIN Person.BusinessEntityAddress ba ON ba.BusinessEntityID = v.BusinessEntityID

JOIN Person.Address a ON a.AddressID = ba.AddressID

UNION

SELECT

s.BusinessEntityID, s.Name, a.City, a.PostalCode, a.AddressLine1

FROM Sales.Store s

JOIN Person.BusinessEntityAddress ba ON ba.BusinessEntityID = s.BusinessEntityID

JOIN Person.Address a ON a.AddressID = ba.AddressID

**UNION w podzapytaniu** → możesz zbudować całe UNION w podzapytaniu i zaaliasować je (podzapytanie, czyli to co jest w nawiasie), po czym wykorzystać w zapytaniu SELECT alias.

Np.

USE AdventureWorks

SELECT

sub.Id, sub.AccountNumber, a.City, a.PostalCode, a.AddressLine1

FROM

(

SELECT c.CustomerID Id, c.AccountNumber

FROM Sales.Customer c

UNION

SELECT v.BusinessEntityID, v.AccountNumber

FROM Purchasing.Vendor v

UNION

SELECT s.BusinessEntityID, s.Name

FROM Sales.Store s

) sub

JOIN Person.BusinessEntityAddress ba ON ba.BusinessEntityID = sub.Id

JOIN Person.Address a ON a.AddressID = ba.AddressID

**Wypełnianie pustej kolumny treścią** → może przydać się korzystając z UNION, że chcąc aby ilość kolumn w zapytaniach SELECT była taka sama jednocześnie chcesz żeby jakaś kolumna, która byłaby pusta (bo w drugim zapytaniu SELECT coś jest) była wypełniona jakąś treścią. Tworzysz tą kolumnę w SELECT i od razu dodajesz wartości, np. 'Seller' AS 'Job' gdzie Job to alias/nazwa kolumny, a Seller to wartość, którą są wypełnione komórki.

USE AdventureWorks

SELECT

pp.LastName, pp.FirstName, 'Seller'

FROM

Sales.SalesPerson sp

JOIN Person.Person pp ON sp.BusinessEntityID = pp.BusinessEntityID

SELECT

pp2.LastName, pp2.FirstName, hre.JobTitle

FROM

HumanResources.Employee hre

JOIN Person.Person pp2 ON hre.BusinessEntityID = pp2.BusinessEntityID

USE AdventureWorks

SELECT

pp.LastName, pp.FirstName, 'Seller' AS 'Job'

FROM

Sales.SalesPerson sp

JOIN Person.Person pp ON sp.BusinessEntityID = pp.BusinessEntityID

UNION

SELECT

pp2.LastName, pp2.FirstName, hre.JobTitle

FROM

HumanResources.Employee hre

JOIN Person.Person pp2 ON hre.BusinessEntityID = pp2.BusinessEntityID

ORDER BY 'Job'

**47. Operator UNION ALL, INTERSECT, EXCERPT**

**Operator UNION ALL, INTERSECT, EXCEPT** → cechą UNION jest to, że jeśli łączone są kolumny w jedną całość, które zawierają rekordy o tych samych identyfikacjach (np. po ID) to one się nie powielają, tylko wykorzystywany jest tylko jeden unikalny rekord. Możesz spojrzeć na działanie tych operatorów jak na zbiory matematyczne (na przykładzie 2 zbiorów: A i B): INTERSECT (wspólne dla A i B), EXCEPT (te, które są unikalne tylko dla A, a których nie ma w B), UNION (wszystko z A i B, ale nie dubluje się wartości) i UNION ALL (wszystko z A i B, ale dubluje się wartości)

**UNION ALL** → biorąc pod uwagę, że UNION pokazuje tylko unikalne rekordy, występujące we wszystkich wybranych kolumnach, jeśli chcę, żeby one dodawały się bez względu na powtarzalność, mogę skorzystać z UNION ALL, które zrobi właśnie to. Operator UNION ALL ponadto mniej obciąża serwer ponieważ nie musi sortować wyników i eliminować zdublowanych wierszy.

Np.

--Zwykłe UNION

USE AdventureWorks

SELECT e.BusinessEntityID

FROM HumanResources.Employee e

UNION

SELECT v.BusinessEntityID

FROM Sales.SalesPerson v

--zwraca 290 rekordów, bo nie wyświetla tych, które się powtarzają

--Zwykłe UNION ALL

USE AdventureWorks

SELECT e.BusinessEntityID

FROM HumanResources.Employee e

UNION ALL

SELECT v.BusinessEntityID

FROM Sales.SalesPerson v

ORDER BY BusinessEntityID

--zwraca 307 rekordów, bo wyświetla też te, które się powtarzają

**Alternatywa dla UNION ALL** → taki sam efekt co UNION ALL można uzyskać wykorzystując zwykłe UNION, ale dla obu łączonych w jedną kolumn dodaj wymyśloną przez ciebie kolumnę, która będzie miała wartości różne od drugiej tabeli. To sprawi, że rekordy dwóch tabel siłą rzeczy będą różne. Zwróć na przykładzie uwagę, że ID się na końcu powtarzają.

Np.

USE AdventureWorks

SELECT e.BusinessEntityID, 'Employee'

FROM HumanResources.Employee e

UNION

SELECT v.BusinessEntityID, 'Salesman'

FROM Sales.SalesPerson v

ORDER BY BusinessEntityID

**INTERSECT** → operator ten bada czy jest część wspólna zbioru skonstruowanego przez wynik pierwszego zapytania ze zbiorem skonstruowanym przez wynik drugiego zapytania.

Np.

USE AdventureWorks

SELECT ProductID

FROM Purchasing.ProductVendor

WHERE BusinessEntityID = 1680

INTERSECT

SELECT

ProductID

FROM Sales.SpecialOfferProduct

**EXCEPT** → można porównać do matematycznej operacji odejmowania zbioru. Wyświetla rekordy, które zostały zwrócone przez pierwsze zapytanie, ale nie przez drugie zapytanie. Nie pokazuje wspólnych rekordów. Jest to odwrotność INTERSECT. Kolejność zapytań SELECT w przypadku EXCEPT jest bardzo istotna.

Np.

USE AdventureWorks

SELECT ProductID

FROM Purchasing.ProductVendor

WHERE BusinessEntityID = 1680

EXCEPT

SELECT

ProductID

FROM Sales.SpecialOfferProduct

**SQL Server - programowanie, typy zaawansowane, XML**

**48. Zmienne w języku TSQL**

**Batch** → zestaw instrukcji SQL złożonych razem i wykonanych jako grupa, jedna po drugiej. Wybieranie linii kodu, które chcesz żeby były wykonane to wybieranie batch-a.

**Poprawne deklarowanie zmiennych w TSQL** → zmienne deklaruje się za pomocą DECLARE oraz SET, ale należy jedno i drugie uruchomić w jednym batch-u żeby zadeklarowanie/przypisanie zadziałało.

Np.

USE AdventureWorks

DECLARE @i INT

SET @i = 332

SELECT \* FROM Production.Product WHERE ProductID = @i

**Alternatywa dla SET i deklaracja wielu zmiennych naraz** → możesz zadeklarować zmienne w jednej linijce bez SET.

Np.

USE AdventureWorks

DECLARE @i INT = 332, @j INT = 505

SELECT \* FROM Production.Product WHERE ProductID = @i OR ProductID = @j

**Przypisywanie wartości zmiennej na podstawie danych z tabeli** → możesz zadeklarować zmienną i przypisać jej typ danych, po czym za pomocą SELECT przypisać jej wartość z tabeli (SELECT zmienna = kolumna FROM Tabela WHERE warunek). Istotną różnicą jest to, że zamieniasz SELECT na SET w tym przypadku i za SELECT kodujesz zwykłe zapytanie FROM/WHERE.

Np.

USE AdventureWorks

DECLARE @i INT = 332

DECLARE @color VARCHAR(50)

SELECT @color = Color FROM Production.Product WHERE ProductID = @i

--PRINT @color

SELECT \* FROM Production.Product WHERE Color = @color

Zadeklarowane zmienne, a uruchamianie kodu w częściach → jeśli spróbujesz uruchomić pojedynczo najpierw kod deklarujący zmienną, a potem pojedynczo kod odwołujący się do tej zmiennej to zwrócony zostanie błąd. Musisz zrobić jedno i drugie naraz.

Np.

USE AdventureWorks

DECLARE @maksymalna INT

SET @maksymalna =

(SELECT MAX(pp.SafetyStockLevel) AS 'Max'

FROM Production.Product pp)

PRINT CAST(@maksymalna AS nvarchar(100))

**49. Instrukcja warunkowa IF**

**IF, ELSE, Begin, End** → instrukcja warunkowa IF sprawdza wynik zapytania i wykonuje określone czynności w zależności od wyniku. Po IF znajdują się BEGIN (wyniki) i END. Są one potrzebne zawsze kiedy wynik IF/ELSE ma więcej niż jedno polecenie, czyli będą potrzebne bardzo często. BEGIN i END zbierają polecenia, które mają zostać wykonane w przypadku spełnienia warunku. ELSE zwraca wynik zapytania, który był inny niż ten w IF, to znaczy jeśli IF nie zostało spełnione.

Np.

USE AdventureWorks

DECLARE @Date DATE = '2016-05-22' --Sunday

IF DATEPART(weekday, @Date) = 1 OR DATEPART(weekday, @Date) = 7

BEGIN

PRINT 'It is a weekend.'

PRINT 'Yay!'

END

ELSE

BEGIN

PRINT 'It is a work day.'

PRINT 'Nay.'

END

**Debugowanie i wykonywanie kodu krok po kroku** → po napisaniu kodu możesz uruchomić debugowanie (przycisk obok Execute) w celu przechodzenia (za pomocą klawisza F10) przez kod i chociażby w instrukcji IF sprawdzanie co się wykonuje (czy IF czy ELSE) i dlaczego.

Np.

USE AdventureWorks

DECLARE @Date DATE = '2016-05-22' --Sunday

IF DATEPART(weekday, @Date) = 1 OR DATEPART(weekday, @Date) = 7

BEGIN

PRINT 'It is a weekend.'

PRINT 'Yay!'

END

ELSE

BEGIN

PRINT 'It is a work day.'

PRINT 'Nay.'

END

--Uruchom debugowanie

**50. Instrukcja warunkowa IF w praktyce - wyznaczanie kursu waluty**

**Rozbudowane zapytanie z IF, nadpisywanie zadeklarowanych wartości zmiennych, PRINT przy SET i brak ELSE** → kolejne instrukcje SET dla tych samych zmiennych zmieniają ich wartości na nowe. Poniższy przykład pokazuje przewalutowanie na podstawie kursu z poprzedniego dnia, dlatego dzień jest odejmowany w obliczeniach i pomijane są weekendy (poniedziałek minus dzień = niedziela, etc.). Nadpisanie zmiennej daty właśnie w celu wykorzystania dnia poprzedniego. Zwróć uwagę, że przy SET jest PRINT - możesz chcieć żeby przy zadeklarowaniu zmiennej pojawił się jakiś tekst, informacja. Jeśli zmienisz datę na sobotę, pierwsze IF (niedziela) nie zostanie wykonane i żadne PRINT się z niego nie pojawi - nie ma w przykładzie ELSE.

Np.

USE AdventureWorks

DECLARE @CourseDate DATE = '2016-05-23' -- Monday

SET @CourseDate = DATEADD(d, -1, @CourseDate)

PRINT 'Yesterday ' + CAST(@CourseDate AS VARCHAR) + ' ' + DATENAME(weekday, @CourseDate) + ' - ' + CAST(DATEPART(weekday, @CourseDate) AS VARCHAR)

IF DATEPART(weekday, @CourseDate) = 1 -- Sunday

BEGIN

PRINT 'It is Sunday. Not a working day.'

SET @CourseDate = DATEADD(d, -1, @CourseDate)

PRINT 'Value is ' + CAST(@CourseDate AS VARCHAR) + ' ' + DATENAME(weekday, @CourseDate) + ' - ' + CAST(DATEPART(weekday, @CourseDate) AS VARCHAR)

END

IF DATEPART(weekday, @CourseDate) = 7 -- Saturday

BEGIN

PRINT 'It is Saturday. Not a working day.'

SET @CourseDate = DATEADD(d, -1, @CourseDate)

PRINT 'Value is ' + CAST(@CourseDate AS VARCHAR) + ' ' + DATENAME(weekday, @CourseDate) + ' - ' + CAST(DATEPART(weekday, @CourseDate) AS VARCHAR)

END

PRINT 'Result is ' + CAST(@CourseDate AS VARCHAR) + ' ' + DATENAME(weekday, @CourseDate) + ' - ' + CAST(DATEPART(weekday, @CourseDate) AS VARCHAR)

**51. Instrukcja warunkowa IF w pracy administratora danych**

**Instrukcja warunkowa IF w pracy administratora danych oraz testy skryptu** → przy pisaniu skryptu tworzy się tabela tymczasowa i wkładane są do niej wartości. Na tym etapie dokonuje się testów skryptu. Tworzenie tabeli tymczasowej za pomocą #tab. Jeśli pojawią się problemy z tabelą tymczasową, a pojawią się w naturalny sposób, nadpisanie #tab za pomocą tego samego batch-a z CREATE TABLE będzie zwracać błąd. Należy na początku skryptu usuwać wcześniejszą wersję tabeli (nadal pozostaje aktualizowany kod, który piszesz pod DROP TABLE) za pomocą DROP TABLE - ale pojawi się błąd jeśli takiej tabeli jeszcze nie będzie.

Np.

IF OBJECT\_ID('tempdb..#tab') IS NOT NULL

BEGIN

PRINT 'Dropping #tab table'

DROP TABLE #tab

END

CREATE TABLE #tab (id int)

INSERT #tab VALUES (1)

**OBJECT\_ID** → w SQL Server każdy obiekt posiada swoją nazwę przyjazną dla administratora/programisty, ale posiada "pod spodem" obiekt ten jest podpisany identyfikatorem - trzeba umieć przekształcać numer identyfikatora na nazwę albo nazwę na numer identyfikatora. Jeśli do OBJECT\_ID dasz w argumencie nazwę nieistniejącego obiektu to zwrócony zostanie NULL. OBJECT\_ID zwraca numer identyfikacyjny obiektu bazy danych o zasięgu schematu.

Np.

IF OBJECT\_ID('tempdb..#tab') IS NOT NULL

BEGIN

PRINT 'Dropping #tab table'

DROP TABLE #tab

END

CREATE TABLE #tab (id int)

INSERT #tab VALUES (1)

SELECT \* FROM #tab

PRINT OBJECT\_ID('tempdb..#tab')

**52. Pętla WHILE**

**Pętla WHILE i NOCOUNT** → pętla WHILE spełniać będzie warunek znajdujący się między BEGIN, a END tak długo, jak spełniony będzie warunek zapisany po WHILE. Wykonywanie pętli WHILE może spowodować pojawienia się wielu informacji w konsoli - jeśli nie chcesz ich widzieć, możesz skorzystać z polecenia SET NOCOUNT ON, które umieścisz przed WHILE i DECLARE.

Np.

CREATE TABLE TestWhile (id int)

SET NOCOUNT ON

DECLARE @i INT = 1

WHILE @i < 1000

BEGIN

INSERT TestWhile VALUES (@i)

SET @i += 1

END

SELECT \* FROM TestWhile

**53. Pętla WHILE w praktyce - podział dużej transakcji kasującej na mniejsze**

**Pętla WHILE w praktyce - podział dużej transakcji kasującej na mniejsze** → na przykładzie widać usuwanie rekordów po kawałku. Robi się tak ponieważ usunięcie wszystkich rekordów naraz przy większych pakietach danych może doprowadzić do błędu. Przykład dla lekcji poniżej.

**@@ROWCOUNT** → w zapytaniu przykładowym wykorzystano polecenie @@ROWCOUNT, które zwraca liczbę wierszy, na które wpływa ostatnia instrukcja. W przykładzie jeśli będzie ona większa od zero wykonywane będzie polecone zadanie, czyli usuwanie kolejnej porcji rekordów.

Np.

CREATE TABLE TestWhile (id int)

SET NOCOUNT OFF

DECLARE @i INT = 1

WHILE @i < 1000

BEGIN

INSERT TestWhile VALUES (@i)

SET @i += 1

END

SELECT \* FROM TestWhile

DELETE TOP(13) FROM TestWhile

WHILE @@ROWCOUNT>0

BEGIN

DELETE TOP(13) FROM TestWhile

END

**Pętla WHILE i WHERE** → chcąc określić które konkretne rekordy mają być usunięte konieczne jest zastosowanie klauzuli WHERE. Patrz przykład dla lekcji.

**BREAK** → instrukcja zatrzymywania pętli. BREAK przerywa działanie pętli bez badania warunku, który normalnie był badany aby stwierdzić czy pętle WHILE należy przerwać. To sposób na awaryjne przerwanie pętli. Patrz przykład dla lekcji.

**WAITFOR DELAY** → blokuje wykonanie partii, procedury składowanej lub transakcji do momentu upłynięcia określonego czasu lub przedziału czasu albo określona instrukcja modyfikuje lub zwraca co najmniej jeden wiersz. Patrz przykład dla lekcji.

**Opisanie przykładu** → SET NOCOUNT OFF wyłącza opcję nie pokazywania obliczeń w konsoli. Deklarowana jako zmienna @StartTime jest teraźniejsza data (czyli momen uruchomienia skryptu). Polecenie usuwa co 13 rekordów tak długo aż ROWCOUNT będzie większy od zera. Pętla WHILE zaczyna się od opóźniania dlaszej części skryptu o sekundę tak długo jak wynik funkcji zwiększającej teraźniejszy moment (zadekalarowana zmienna) + 10 sekund będzie mniejszy niż GETDATE(). Innymi słowy zmienna zostałą zadeklarowana wcześniej, poza pętlą i się nie zmieni. W pętli jedynie za każdym razem, za każdą iteracją ma dodane te 10 sekund, więc za każdym razem jest taka sama. Jednocześnie GETDATE() do którego jest porównywana zmienai się co sekundę (bo takie jest wprowadzone opóźnienie na początku pętli) i po upływie 10 sekund będzie w końcu większe i uruchomi BREAK. Natomiast za kulisami, przez te 10 sekund wykonuje się reszta, nie ruszona przez BREAK czyli usuwanie co 13 rekordów.

Np.

CREATE TABLE TestWhile (id int)

SET NOCOUNT OFF

DECLARE @i INT = 1

WHILE @i < 1000

BEGIN

INSERT TestWhile VALUES (@i)

SET @i += 1

END

DECLARE @StartTime DATETIME = GETDATE()

DELETE TOP (13) FROM TestWhile

WHILE @@ROWCOUNT>0

BEGIN

WAITFOR DELAY '00:00:01'

IF DATEADD(s, 10, @StartTime) < GETDATE()

BEGIN

PRINT 'Breaking loop - time exceeded...'

BREAK

END

DELETE TOP(13) FROM TestWhile

END

**WHILE i WHERE** → jeśli chcesz dodawać warunki do pętli WHILE, możesz to zrobić za pomocą WHERE umieszczonego w typowym dla siebie miejscu.

Np.

DECLARE @Id INT = 1, @Num INT = 1000, @Let\_1 NVARCHAR(1) = 'a', @Let\_2 NVARCHAR(1) = 'b', @Let\_3 NVARCHAR(1) = 'c', @Date DATETIME2 = GETDATE()

CREATE TABLE #TableForTwo(Id INT, Numbers INT, Letters NVARCHAR(1), [Date] DATETIME2)

WHILE @Id <= 333

BEGIN

INSERT INTO #TableForTwo ([Id], [Letters], [Numbers], [Date])

VALUES (@Id, @Let\_1, @Num, @Date)

SET @Id += 1

END

SET @Num = 1001

WHILE @Id > 333 AND @Id <= 666

BEGIN

INSERT INTO #TableForTwo ([Id], [Letters], [Numbers], [Date])

VALUES (@Id, @Let\_2, @Num, @Date)

SET @Id += 1

END

SET @Num = 1002

WHILE @Id > 666 AND @Id <= 1000

BEGIN

INSERT INTO #TableForTwo ([Id], [Letters], [Numbers], [Date])

VALUES (@Id, @Let\_3, @Num, @Date)

SET @Id += 1

END

SELECT \* FROM #TableForTwo

--Usuwanie tabeli przykładowej

DROP TABLE #TableForTwo

--Usuwanie po 50 wierszy z WHERE

WHILE @@ROWCOUNT > 0

BEGIN

DELETE TOP (50) FROM #TableForTwo WHERE Letters = 'b'

END

**54. Typ ROWVERSION**

**Typ ROWVERSION** → w przykładzie wykorzystano bazę danych tempdb, a nie AdventureWorks. Zwróć uwagę, że każdy UPDATE i każdy INSERT wykonywany powoduje, że automatycznie i bez żadnej dodatkowej ingerencji, numer zapisany w kolumnie ver (typ ROWVERSION) będzie się zwiększał. Ponadto liczba ta obowiązuje na całą bazę danych, tzn. nie jest specyficzna dla określonej tabeli - jeśli w bazie danych znajduje się kilka tabel wykorzystujących ROWVERSION to numery nadawane przez ROWVERSION zachowają ciągłość przez wszystkie te tabele. Liczba ta jest wspólna dla wszystkich tabel. Wstawiając wartość do tabeli przez INSERT nie bierzesz pod uwagę wartości ROWVERSION, bo ta jest wstawiana automatycznie. To co wstawiłeś w INSERT ma nadany numer z ROWVERSION, ale zmieni się on gdy zrobisz UPDATE rekordu.

Np

USE tempdb

CREATE TABLE #Test (Name NVARCHAR(30), Ver ROWVERSION)

INSERT #Test(Name) VALUES ('Start')

SELECT \* FROM #Test

INSERT #Test(Name) VALUES ('Second')

SELECT \* FROM #Test

UPDATE #Test SET Name = 'First' WHERE Name = 'Start'

SELECT \* FROM #Test

CREATE TABLE #Test2 (Name NVARCHAR(30), Ver ROWVERSION)

INSERT #Test2(Name) VALUES ('Start')

SELECT \* FROM #Test2

INSERT #Test2(Name) VALUES ('Second')

SELECT \* FROM #Test2

UPDATE #Test2 SET Name = 'First' WHERE Name = 'Start'

SELECT \* FROM #Test2

INSERT #Test(Name) VALUES ('Third')

SELECT \* FROM #Test

**Praktyczne zastosowanie ROWVERSION (optymistyczne lokowanie)** → optymistyczne lokowanie polega na założeniu, że gdy inni użytkownicy mają dostęp do rekordów to ich nie zmieniają. Można to sprawdzić pisząc zapytanie UPDATE dla rekordów, które nas interesują - dodając w zapytaniu dodatkowy warunek WHERE sprawdzający czy wersja jest równa znanej, oryginalnej wersji rekordu. Należy dodać jeszcze SELECT @@ROWCOUNT, która zwraca liczbę wierszy na które wpłynęła ostatnia instrukcja. Jeśli wynik będzie (dla jednego rekordu, jak w przykładzie) 1 to oznacza, że nikt nic nie zmieniał, bo ja właśnie mogłem coś zmienić. Wynik 0 oznacza, że zmiany były wprowadzane, bo oryginalne wersje już nie istnieją. Na przykładzie raz uruchomisz apytanie będzie 1, a drugi raz 0 bo oczywiście za pierwszym razem coś zmieniłeś. Można dodać komentarz w przypadku 0 dla innych użytkowników informując ich, że dokonywane były zmiany.

Np.

SELECT \* FROM #Test WHERE Name = 'First'

UPDATE #Test SET Name = 'FIRST' WHERE Name = 'First' AND Ver = 0x00000000000007D3

SELECT @@ROWCOUNT

**55. Typ UNIQUEIDENTIFIER**

**Typ UNIQUEIDENTIFIER** → to unikalny typ danych w SQL. Nie jest używany żeby pokazać wartość w kolumnie, ale ze względów technicznych. Ten typ danych zapewnia chociażby to, że jeśli baza danych jest replikowana między różnymi serwerami, a użytkownicy tworzą kolejne zmienne, niezależnie od siebie to wykorzystując uniqueidentifier zapewniam bardzo wysokie prawdopodobieństwo, że rekordy nie będą się powtarzać (nawet jeśli będą miały te same wartości zmiennych). Do tworzenia tego typu danych c zawsze trzeba wykorzystywać funkcję NEWID.

**NEWID** → funkcja niezbędna do tworzenia typu zmiennych jakim jest uniqueidentifier. Każdy identyfikator to wartość 16-bajtowa. Jest to algorytm który gwarantuje losowość tworzonej zmiennej. Zwróć uwagę, że kolejność tworzonych ID też jest losowa jeśli je posortować.

Np.

USE tempdb

CREATE TABLE TestGUID

(

ID uniqueidentifier

,Name NVARCHAR(100)

)

INSERT TestGUID VALUES (NEWID(), 'One'), (NEWID(), 'Two'), (NEWID(), 'Three')

SELECT \* FROM TestGUID ORDER BY ID

**NEWSEQUENTIALID** → korzystając z uniqueidentifier zapewniasz sobie losowość, ale nie zapewnisz tego żeby ID następowały jedna o drugiej, tak jak 1, 2, 3, 4, 5... Jest to przydatne bo tak działa logika zwykłych ID i zapewnia to porządek. Żeby go zapewnić skorzystaj z NEWSEQUENTIALID. Ta funkcja musi być zdefiniowana jako DEFAULT dla uniqueidentifier.

Np.

USE tempdb

CREATE TABLE TestGUIDSequential

(

ID uniqueidentifier DEFAULT NEWSEQUENTIALID()

,Name NVARCHAR(100)

)

INSERT TestGUIDSequential(Name) VALUES ('One'), ('Two'), ('Three')

SELECT \* FROM TestGUIDSequential ORDER BY ID

**Konwertowanie uniqueidentifier na tekst** → możesz chcieć zadeklarować zmienną typu uniqueidentifier jako tekst, a nie liczbę. Możesz dokonać konwersji poprzez polecenie CAST.

Np.

DECLARE @u uniqueidentifier = NEWID()

SELECT @u, CAST(@u AS VARCHAR(100)) AS 'char'

**Przypisywanie uniqueid do wartości istniejącej tabeli** → za pomocą JOIN.

Np.

CREATE TABLE #TestTable

(ID int, Test uniqueidentifier)

DECLARE @i int = 1

WHILE @i <= 100

BEGIN

INSERT INTO #TestTable([ID], [Test]) VALUES (@i, NEWID())

SET @i += 1

END

SELECT \* FROM #TestTable

USE AdventureWorks

SELECT

pp.BusinessEntityID

,pp.FirstName + '' + pp.MiddleName + '' + pp.LastName AS 'Full Name'

,tt.ID

,tt.Test

FROM Person.Person pp

JOIN #TestTable tt ON pp.BusinessEntityID = tt.ID

**56. Collation - co to jest collation?**

**Collation** → odpowiada za sortowanie napisów. SQL traktuje a < ą, czyli a jest wcześniejsza od ą. Jeśli collation jest niewłaściwe, to niedobrze będą sortowane teksty, w efekcie na chociażby liście alfabetycznej nazwisk zobaczysz wszystkie osoby o "normalnych" nazwiskach, a na końcu z nazwiskami na "Ą", "Ę"...

**SERVERPROPERTY** → to zapytanie sprawdza collation dla serwera. Collation znajduje się również we właściwościach (Properties) poprzez wejście do właściwości serwera.

Np.

USE AdventureWorks

SELECT SERVERPROPERTY('COLLATION')

**DATABASEPROPERTYEX** → sprawdza collation dla danej bazy danych. Inaczej możesz też wejść do właściwości bazy danych żeby znaleźć to konkretne collation.

Np.

USE AdventureWorks

SELECT DATABASEPROPERTYEX('AdventureWorks', 'COLLATION')

**Przedefiniowanie collation** → poprzez wejście w Properties-Options bazy danych. Nie oznacza to, że już istniejące tabele będą używały nowego collation. Wszystkie obiekty trzeba na nowo przebudować, a zatem usunąć tabele i stworzyć je od nowa.

**Podział collation** → dzieli się je na SQL-owe i windows. SQL-owe są zaszłością i ich nazwy zaczynają się od "SQL\_". Pozostałe (windows) zwykle zaczynają się od nazwy krajów gdzie to sortowanie zobowiązuje.

**Definiowanie collation z dokładnością do pola tekstowego** → wybierz polecenie design na tabeli w eksploratorze tabel i naciśnij na jedno z pól tekstowych. W column properties znajdziesz collation. Może być ustawione na <database default>, a zatem zgodne z tym co jest w bazie danych.

**Praktyczne zastosowanie collation** → w przykładzie zastosowano NVARCHAR, a zatem nie będzie problemu ze znakami narodowymi. Zapis CI w collation = Case Insensitive, a zatem nie SQL nie zwróci uwagi na wielkość liter. CS = Case Sensitive, a zatem odwrotność poprzedniego, zwraca uwagę. AS = Accent Sensitive, istnieją języki gdzie słowo wypowiedizane z akcentem w jednym miejscu oznacza coś innego jakby było w innym miejscu - ustawienie to nie dotyczy języka polskiego.

Np.

CREATE TABLE #Test1

(

Name\_test1 NVARCHAR(100) COLLATE Latin1\_General\_CI\_AS,

Name\_test2 NVARCHAR(100) COLLATE Latin1\_General\_CS\_AS

)

INSERT #Test VALUES ('Berlin', 'BERLIN')

INSERT #Test VALUES ('San Francisco', 'SAN FRANCISCO')

SELECT \* FROM #Test WHERE Name\_test1 = 'Berlin'

SELECT \* FROM #Test WHERE Name\_test1 = 'BERLIN'

SELECT \* FROM #Test WHERE Name\_test2 = 'Berlin'

SELECT \* FROM #Test WHERE Name\_test2 = 'BERLIN'

**Collation conflict** → jeśli spróbujesz łączyć w jakikolwiek sposób dwie zmienne o różnych collation, pojawi się błąd. Rozwiązaniem jest dodanie do zapytania w jaki sposób ma się odbywać porównywanie napisów tekstowych. To uszczególnienie na skalę pojedynczego warunku klauzuli WHERE. Za klauzulą WHERE piszesz o jaką operację ci chodzi COLLATE wybrane collate.

Np.

CREATE TABLE #Test2

(

Name\_test1 NVARCHAR(100) COLLATE Latin1\_General\_CI\_AS,

Name\_test2 NVARCHAR(100) COLLATE Latin1\_General\_CS\_AS

)

INSERT #Test2 VALUES ('Berlin', 'BERLIN')

INSERT #Test2 VALUES ('San Francisco', 'SAN FRANCISCO')

--Wielkość liter nie ma znaczenia

SELECT \* FROM #Test2 WHERE Name\_test1 = Name\_test2 COLLATE Latin1\_General\_CI\_AS

--Wielkość liter ma znaczenie

SELECT \* FROM #Test2 WHERE Name\_test1 = Name\_test2 COLLATE Latin1\_General\_CS\_AS

**Wyświetlanie listy collations w wynikach zapytań** → za pomocą SELECT \* FROM sys.fn\_helpcollations().

Np.

SELECT \* FROM sys.fn\_helpcollations()

**57. Collation - literał N przed stałymi napisowymi**

**Collation - literał N przed stałymi napisowymi** → stawianie litery N przed apostrofem przy deklarowaniu wartości oznacza, że litera wewnątrz apostrofu ma być traktowana jako UNICODE.

**CREATE DATABASE** → funkcja tworząca bazę danych.

Np.

CREATE DATABASE Test\_NO\_Pl

**DROP DATABASE** → funkcja usuwająca bazę danych.

Np.

CREATE DATABASE Test\_NO\_Pl

DROP DATABASE Test\_NO\_Pl

**TRUNCATE TABLE** → to funkcja usuwająca dane wewnątrz tabeli, ale nie samą tabelę.

Np.

CREATE DATABASE Test\_NO\_Pl COLLATE Latin1\_General\_CI\_AS

USE Test\_NO\_Pl

CREATE TABLE Test(txt NVARCHAR(100))

INSERT Test VALUES (N'A'), (N'Ą'), (N'B'), (N'C'), (N'Ć'), (N'D')

SELECT \* FROM Test ORDER BY txt

TRUNCATE TABLE Test

**Uruchamianie tabeli z polskimi znakami i nieodpowiednim COLLATE w bazie danych** → polskie specjalne znaki zostaną zignorowane.

Np.

USE Test\_NO\_Pl

CREATE TABLE Test(txt NVARCHAR(100))

INSERT Test VALUES ('A'), ('Ą'), ('B'), ('C'), ('Ć'), ('D')

SELECT \* FROM Test ORDER BY txt

TRUNCATE TABLE Test

**COLLATION polskie** → COLLATION dostosowane do polskich użytkowników do Polish\_CI\_AS

**58. Typ HIERARCHYID - wprowadzanie danych**

**Typ HIERARCHYID - wprowadzanie danych** → aplikacja może przechowywać dane w sposób hierarchiczny. Należy dodać, że obsługa tego typu danych nie jest intuicyjna. W przykładzie typ danych hierarchyid jest kluczem podstawowym tabeli.  Szczyt hierarchii określony został jako zero. Przykład poniżej.

**hierarchyID** → Typ danych hierarchyid to systemowy typ danych o zmiennej długości. Użyj hierarchyid do reprezentowania pozycji w hierarchii. Kolumna typu hierarchyid nie reprezentuje automatycznie drzewa. Do aplikacji należy wygenerowanie i przypisanie wartości hierarchyid w taki sposób, aby pożądana relacja między wierszami znalazła odzwierciedlenie w wartościach. Przykład poniżej.

**GetLevel** → zwraca liczbę całkowitą reprezentującą głębokość węzła w drzewie. Przykład poniżej.

**Indeks i indeks zgrupowany** → indeks to struktura na dysku powiązana z tabelą lub widokiem, która przyspiesza pobieranie wierszy z tabeli lub widoku. Indeks zawiera klucze utworzone na podstawie jednej lub większej liczby kolumn tabeli lub widoku. Tabela lub widok może zawierać typ indeksu zgrupowanego. Indeksy klastrowe (CLUSTERED) sortują i przechowują wiersze danych w tabeli lub widoku na podstawie ich kluczowych wartości. Te wartości kluczy to kolumny zawarte w definicji indeksu. Na tabelę może przypadać tylko jeden indeks klastrowy, ponieważ same wiersze danych można przechowywać tylko w jednej kolejności. Jedynym przypadkiem, w którym wiersze danych w tabeli są przechowywane w kolejności posortowanej, jest sytuacja, gdy tabela zawiera indeks klastrowy. Jeśli tabela ma indeks klastrowy, nazywa się ją tabelą klastrową. Jeśli tabela nie ma indeksu klastrowego, jej wiersze danych są przechowywane w nieuporządkowanej strukturze zwanej stertą. Przykład poniżej.

**UNIQUE** → Ograniczenie UNIQUE zapewnia, że wszystkie wartości w kolumnie są różne. Zarówno ograniczenia UNIQUE, jak i PRIMARY KEY zapewniają gwarancję niepowtarzalności kolumny lub zestawu kolumn. Ograniczenie KLUCZ PODSTAWOWY automatycznie ma ograniczenie UNIKALNE. Można jednak mieć wiele ograniczeń UNIQUE na tabelę, ale tylko jedno ograniczenie PRIMARY KEY na tabelę. Przykład poniżej.

**Budowanie indeksów** → dwa sposoby budowania indeksów. Pierwszy to breadth first. Ten typ indeksu należy stosować kiedy moje zapytania będą sprowadzać się do wyciągania wszystkich rekordów znajdujących się na tym samym poziomie. Tworzenie indeksu zależy od kolejności wprowadzania kolumn w klauzuli ON - w tym przypadku pierwsza będzie kolumna określająca poziom zmiennej. Drugi rodzaj indeksu to depth first: tutaj jako druga, a nie pierwsza kolumna będzie odpowiedzialna za poziom - ten indeks będzie lepiej pasował zapytaniom, które będą zwracać wartość i wszystkie poniższe poziomem wartości. Przykład poniżej.

**Wstawianie rekordu do hierarchicznej tabeli** → za pomocą INSERT i VALUES, ale OrgNode inicjowany jest w taki sposób, że inicjujemy metodę statyczną GetRoot na rzecz hierarchyid. Taki zapis hierarchyid::GetRoot() zwraca identyfikator, który stanie się pierwszym elementem hierarchii. Nie jest wstawiany ponadto OrgLevel, bo jest to wartość wyliczana i nie można jej określać podczas wstawiania rekordów. Przykład poniżej.

**::GetRoot** → zwraca korzeń drzewa hierarchii. GetRoot() jest metodą statyczną. Przykład poniżej.

**ToString** → zwraca ciąg znaków z logiczną reprezentacją tego. ToString jest wywoływany niejawnie, gdy następuje konwersja z hierarchyid na typ ciągu. Działa jako przeciwieństwo Parse (silnik bazy danych). Przykład poniżej.

**ToString** → OrgNode bez konwertowania na tekst nic ci nie powie. Dlatego podczas wyświetlania najczęściej stosować będziesz tą metodę - dowiesz się jaki tekst odpowiada za ten element. Przykład poniżej.

**Wstawianie rekordu niższego w hierarchii** → żeby wstawić taki rekord trzeba znać identyfikator rodzica, dlatego na początku definiuje się zmienną, chociażby @Manager typu hierarchyid i za pomocą polecenia SELECT wyszukuję rekord odpowiadający za główny element hierarchii. Wstawianie wartości do nowego rekordu wygląda tak, że do zwykłych kolumn wstawia się normalnie w INSERT VALUES. Natomiast OrgNode trzeba wyliczyć i żeby to zrobić potrzebna jest metoda GetDescendant. Przykład poniżej.

**GetDescendant i hierarchia pozioma** → tworząc i dodając rekordy do tabeli hierarchicznej możesz nie tylko hierarchię tą określać pionowo, ale też poziomo (na tym samym poziomie). Wyobraź sobie drzewo genealogiczne i to kto jest ważniejszy zależy od tego czy jest starszy czy młodszy. Metoda GetDescendant w argumentach przyjmuje czy po lewej lub prawej stronie znajdują się inne rekordy. Argumenty (NULL, NULL) określają, że nie ma żadnego rekordu po lewej ani prawej stronie. Gdyby chcieć dodać element po prawej stronie, a już jakieś elementy istniały to pierwszy argument powinien odpowiadać maksymalnemu obiektowi potomnemu. Przykład poniżej.

**CREATE PROC** → Procedura dodawania rekordów do tabeli. Dodawanie rekordów jest problematyczne. Procedura zamknięta jest z poleceniu BEGIN-END. Idąc za przykładem Microsoft można utworzyć procedurę, która będzie dodawać nowe rekordy do tabeli. Procedura na początku otrzymuje identyfikator obiektu nadrzędnego (przykład: @mgrid), nowo wstawionego rekordu (@empid), nazwę i tytuł. Polecenie INSERT na rzecz rodzica wywoła polecenie GetDescendant, ale nowo dodawany rekord będzie dodawać po prawej stronie jako “najmłodsze dziecko" w związku z czym trzeba będzie znaleźć identyfikator "ostatniego dziecka". Najpierw trzeba wyznaczyć wartości dla tych zmiennych. Odnalezienie identyfikatora nadrzędnego: wykona to zapytanie SELECT @mOrgNode = OrgNode... (co oznacza "Odnajdź mi obiekt hierarchii dla "rodzica"). Następnie należy znaleźć obiekt "potomny" znajdujący się maksymalnie po prawej stronie. Całą operację przeprowadzi się w transakcji TRANSACTION ISOLATION LEVEL SERIALIZABLE to znaczy, że w danej chwili na tych obiektach będzie mogła pracować tylko jedna transakcja. Polecenie SELECT w transakcji zwraca "dziecko" obiektu nadrzędnego z największym identyfikatorem. COMMIT to polecenie SQL używane do przechowywania zmian dokonanych w transakcji. Dodawanie rekordów nie będzie skomplikowane ponieważ należy wywoływać utworzoną procedurę INSERT VALUES. Kolejne rekordy dodawane są poprzez EXEC procedura i podanie wartości zmiennych. Zauważ, że Text\_OrgNode Admirała to /1/, a wiceadmirała /2/1/, bo to drugi stopień. Przykład poniżej.

Np.

USE tempdb

CREATE TABLE dbo.EmployeeOrg

(

OrgNode hierarchyid PRIMARY KEY CLUSTERED,

OrgLevel AS OrgNode.GetLevel(),

EmployeeID int UNIQUE NOT NULL,

EmpName varchar(20) NOT NULL,

Title varchar(20) NULL

);

CREATE UNIQUE INDEX EmployeeOrgNc1

ON dbo.EmployeeOrg(OrgLevel, OrgNode)

INSERT dbo.EmployeeOrg (OrgNode, EmployeeID, EmpName, Title)

VALUES (hierarchyid::GetRoot(), 6, 'David', 'Fleet Admiral')

SELECT OrgNode.ToString() AS Text\_OrgNode,

OrgNode, OrgLevel, EmployeeID, EmpName, Title

FROM dbo.EmployeeOrg

DECLARE @Manager hierarchyid

SELECT @Manager = hierarchyid::GetRoot()

FROM dbo.EmployeeOrg;

INSERT dbo.EmployeeOrg (OrgNode, EmployeeID, EmpName, Title)

VALUES (@Manager.GetDescendant(NULL, NULL), 46, 'John', 'Admiral 1')

SELECT OrgNode.ToString() AS Text\_OrgNode,

OrgNode, OrgLevel, EmployeeID, EmpName, Title

FROM dbo.EmployeeOrg

CREATE PROC AddEmp(@mgrid int, @empid int, @e\_name varchar(20), @title varchar(20))

AS

BEGIN

DECLARE @mOrgNode hierarchyid, @lc hierarchyid

SELECT @mOrgNode = OrgNode

FROM dbo.EmployeeOrg

WHERE EmployeeID = @mgrid

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION

SELECT @lc = max(OrgNode)

FROM dbo.EmployeeOrg

WHERE OrgNode.GetAncestor(1) = @mOrgNode;

INSERT dbo.EmployeeOrg (OrgNode, EmployeeID, EmpName, Title)

VALUES (@mOrgNode.GetDescendant(@lc, NULL), @empid, @e\_name, @title)

COMMIT

END;

EXEC AddEmp 6, 271, 'Steven', 'Admiral 2' ;

EXEC AddEmp 6, 119, 'Ken', 'Admiral 3' ;

EXEC AddEmp 46, 269, 'Michael', 'Vice Admiral 1' ;

EXEC AddEmp 271, 272, 'Greg', 'Vice Admiral 2' ;

SELECT OrgNode.ToString() AS Text\_OrgNode,

OrgNode, OrgLevel, EmployeeID, EmpName, Title

FROM dbo.EmployeeOrg ;

**(...)**

**65. XML - wprowadzenie // 66. XML - namespace**

**XML** → kod HTML jest podobny do XML. Pliki XML to pliki tekstowe, które łatwo jest skopiować, przenieść i udostępnić, wysłać mailem, umieścić na stronie www, itp. Nawet przy użyciu notatnika struktura pliku XML powinna być czytelna. Plik ten zwykle na początku ma w nagłówku wersję pliku xml. Potem są tagi rozpoczynające i  kończące dany zbiór informacji. Znajduje się również element główny i prawidłowo powinien być tylko jeden - nazywa się to root element. Dalej są elementy i podelementy. Można zapytać czemu w określonych sytuacjach nie korzystać z CSV (comma seperated values) - po to, że każda organizacja będzie w stanie odczytać plik XML. Pracując z XML korzystasz z bibliotek, które pozwalają na wczytywanie i przekształcanie tego typu plików. Podział na pliki XML : document i fragment. Dokument posiada nagłówek (root), a fragment charakteryzuje się tym, że elementy nie mają jednego głównego nagłówka spinającego je razem.

**XML - namespace** → w języku HTML można tworzyć tabele za pomocą <table>, a zamykasz je za pomocą </table>. Definicja każdego wiersza tabeli zaczyna się od taga <tr> i kończy </tr>. Jeśli chcesz zdefiniować wartości tabeli to korzystasz z <td> i zamykasz </td>. Jeśli jednak np. będziesz wstawiać przedmiot "stół" czyli <table> to wejdzie to w konflikt języka HTML, który przez <table> rozumie tabelę. Możesz tego uniknąć tworząc znaczniki: <h: table> (też "h:" każdym elementem) to tabela, a <f:table> to mebel. Przy publikowaniu danych w postaci XML, trzeba określić jakie słowa będą mogły być używane jako znaczniki w plikach XML. Taka specyfikacja powinna posiadać swoja nazwę, unikalny identyfikator, czyli Unique Resource Indentifier, czyli URI. Dobrą praktyką jest jako URI ustawić adres strony, która publikuje plik XML (to nie musi być nawet istniejąca strona). Np. <r:table xmls:h="http://www.w3.org/TR/html14/">. Tym jest namespace: zbiór słów których wolno użyć podczas definiowania dokumentu XML.

**Problem XML** → jeśli jakiś dokument powołuje się na namespace to nie da się dokumentu odczytać bez powoływania się na namespace. Nieważne, że nie zna się specyfikacji tego namespace, nie wie się jakie znaczniki gdzie mają się znajdować. Ważne żeby funkcja powołująca się na dany XML udawała, że zna tą specyfikację. Sprawdzanie XML dokonuje się za pomocą schematów XML.

**Korzystanie z XML z informacjami z różnych specyfikacji** → jeśli korzystasz z takiego pliku to wszystkie namespace zostaną wymienione w głównym elemencie root. Wzór: <root xmlns:h="http://www.w3.org/TR/html14/" xmlns:f="http://www.mysuperfurniturefactory.com/catalog1". Literki h i f w przykładzie pozwalają odwoływać się w kodzie XML do elementów i atrybutów występujących w dokumencie w skrócony sposób.

**Znacznik** → Znaczniki XML wyglądają podobnie do znaczników HTML, tzn. znacznik startowy rozpoczyna się od znaku "<", a znacznik końcowy od "</". Oba kończą się znakiem ">". Pomiędzy tymi znakami znajduje się nazwa znacznika, która może być dowolnie definiowana. Powinna ona odnosić się i opisywać zawartość znacznika, a nie sposób jego prezentacji.

Np.

 <rozklad>

   <miejscowosc>Poznań</miejscowosc>

   <linia>

    <numer>5</numer>

    <poczatek>Górczyn</poczatek>

    <koniec>Miłostowo</koniec>

   </linia>

   <linia>

    <numer>105</numer>

    <poczatek>Rondo Rataje</poczatek>

    <koniec>Piątkowo</koniec>

   </linia>

 </rozklad>

**Atrybut** →  Znaczniki XML mogą posiadać atrybuty. Jest to para nazwa-wartość, która jest powiązana ze znacznikiem startowym elementu. Nazwy są oddzielone od wartości znakiem "=". Wartość powinna być otoczona pojedynczym lub podwójnym apostrofem. Częstym pytaniem jest pytanie o to czy daną informację powinniśmy zapisywać jako atrybut, czy może jako element podrzędny. Niestety nie ma prostej i jednoznacznej odpowiedzi na tego typu pytania. Przyjmuje się, że atrybuty służą do zapisywania meta-informacji, natomiast elementy podrzędne wykorzystywane powinny być do przechowywania informacji właściwej. Problem polega jednak na tym, że czasem trudno jest ustalić co jest informacją właściwą, a co meta-informacją. Istnieją jednak ograniczenia, które mogą być pewną wskazówką w tej kwestii. Po pierwsze dany element nie może mieć dwóch lub więcej atrybutów o tej samej nazwie. Po drugie, atrybuty mogą przyjmować tylko wartości w postaci ciągu znaków, co jest w wielu sytuacjach poważnym ograniczeniem. Przykład:

Np.

 <rozklad okres="letni">

   <miejscowosc>Poznań</miejscowosc>

   <linia status="remontowana" typ="tramwaj">

    <numer>5</numer>

    <poczatek>Górczyn</poczatek>

    <koniec>Miłostowo</koniec>

   </linia>

   <linia typ="autobus">

    <numer>105</numer>

    <poczatek>Rondo Rataje</poczatek>

    <koniec>Piątkowo</koniec>

   </linia>

 </rozklad>

**Element** → Element XML to wszystko, od (włącznie) znacznika początkowego elementu do (włącznie) znacznika końcowego elementu. Element może zawierać: tekst, atrybuty, inne elementy lub mieszankę powyższych.

Np.

<cena>29,99</cena>

**67. XML - eksport danych z wykorzystaniem FOR XML RAW**

**Tworzenie fragmentu XML z poziomu SQL Server** → przygotuj odpowiednie zapytanie SELECT, wyświetlające dane, które ciebie interesują. Na koniec dodaj FOR XML RAW. Pojawia się odnośnik - naciśnij na niego. To co powstało to nie dokument tylko fragment. Dokument miałby element root.

Np.

USE AdventureWorks

SELECT

    c.CustomerID, c.AccountNumber

    ,h.SalesOrderID, h.OrderDate, h.Status

    ,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML RAW

**Tworzenie pliku XML z poziomu SQL Server** → dodaj na końcu zapytania SELECT: FOR XML RAW('OrderInfo'), ROOT('Orders'). Teraz każdy wiersz ma nazwę elementu nie domyślną, ale "OrderInfo". Dodajesz element ROOT i nadajesz mu nazwę.

Np.

USE AdventureWorks

SELECT

    c.CustomerID, c.AccountNumber

    ,h.SalesOrderID, h.OrderDate, h.Status

    ,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML RAW('OrderInfo'), ROOT('Orders')

**Tworzenie pliku XML, a NULL** → wartości NULL są w procesie tworzenia pliku XML opuszczane. Aby zmusić SQL Server żeby dodał informacje o wartościach NULL należy dodać fragment ELEMENTS XSINIL. Powstanie w elemencie Orders odwołanie do namespace-a, który określa co to jest NULL. W języku XML nie ma NULL więc jest zastąpiony aliasem przypisanym do tego namespace-a = "true".

Np.

USE AdventureWorks

SELECT

    c.CustomerID, c.AccountNumber

    ,h.SalesOrderID, h.OrderDate, h.Status, h.PurchaseOrderNumber

    ,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML RAW('OrderInfo'), ROOT('Orders'), ELEMENTS XSINIL

**Tworzenie pliku XML, którego obiekty są opisane nie przy użyciu atrybutów, ale elementów** → wcześniejsze przykłady pokazywały plik XML jako zbiór atrybutów przypisanych do ROOT (wiersz = atrybut). Można przedstawić ten plik jako zbiór elementów, gdzie każda wartość znajdzie się między rozpoczęciem, a zakończeniem elementu. Należy to zrobić dodając funkcję ELEMENTS na końcu instrukcji FOR.

Np.

USE AdventureWorks

SELECT

    c.CustomerID, c.AccountNumber

    ,h.SalesOrderID, h.OrderDate, h.Status, h.PurchaseOrderNumber

    ,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML RAW('OrderInfo'), ROOT('Orders'), ELEMENTS

**68. XML - eksport danych z wykorzystaniem FOR XML AUTO**

**XML, eksport danych z wykorzystaniem FOR XML AUTO** → jeśli nie chcesz żeby informacje, które uzyskujesz w ramach tworzenia pliku XML powtarzały się (np. jest wiele wierszy dla jednego CustomerID) tylko były one prezentowane razem skorzystasz z FOR XML AUTO. Teraz kod ma postać elementów, w których zagnieżdżone są podelementy. Wadę tego rozwiązania jest, że jest bardzo wrażliwe na zapis kodu SQL SELECT. Jeśli np. posortujesz nie za pomocą głównego klucza, czyli chociażby ID, a czegoś innego, to głównych elementów będzie tyle ile unikalnych podelementów tego "czegoś". Kolejność kolumn też jest istotna - weź pod uwagę, że kolejność kolumn powinna być logiczna. Tak samo jest z JOIN.

Np.

USE AdventureWorks

SELECT

c.CustomerID, c.AccountNumber

,h.SalesOrderID, h.OrderDate, h.Status, h.PurchaseOrderNumber

,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML AUTO, ROOT ('Orders'), ELEMENTS XSINIL

**69. XML - eksport danych z wykorzystaniem FOR XML PATH**

**XML, eksport danych z wykorzystaniem FOR XML PATH** → tworzenie pliku XML gdzie każdy element prezentowany jest wewnątrz tagu <row>. Nadaj mu nazwę (FOR XML PATH('Nazwa'), dodaj ROOT aby stworzyć dokument (FOR XML PATH('Nazwa'), ROOT('ORDERS'). W przypadku FOR XML PATH ode mnie zależy jak będzie wyglądać dokument XML. W przykładzie kolumnom zostały nadane aliasy ('@alias', które znalazły się w kodzie XML. Aliasując na zasadzie 'element/@atrybut' (np. h.OrderDate AS 'Details/@Date', h.Status AS 'Details/@Status') pojawi się element Details, który posiada dwa atrybuty: Date i Status.

Np.

USE AdventureWorks

SELECT

c.CustomerID AS '@CustID', c.AccountNumber AS '@AccNum'

,h.SalesOrderID, h.OrderDate AS 'Details/@Date', h.Status AS 'Details/@Status', h.PurchaseOrderNumber

,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML PATH ('Order'), ROOT ('ORDERS')

**Wykorzystanie @** → składnia @ oznacza, że dana kolumna stanie się atrybutem, a jej brak, że kolumna stanie się elementem.

Np.

USE AdventureWorks

SELECT

c.CustomerID AS '@CustID', c.AccountNumber AS '@AccNum'

,h.SalesOrderID, h.OrderDate AS 'Details/Date', h.Status AS 'Details/Status', h.PurchaseOrderNumber

,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML PATH ('Order'), ROOT ('ORDERS')

**Dodawanie elementów wartości NULL** → jak poprzednio, dodając ELEMENTS XSINIL. W miejscu gdzie jest null podana zostanie informacja: xsi:nil="true".

Np.

USE AdventureWorks

SELECT

c.CustomerID AS '@CustID', c.AccountNumber AS '@AccNum'

,h.SalesOrderID, h.OrderDate AS 'Details/Date', h.Status AS 'Details/Status', h.PurchaseOrderNumber

,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML PATH ('Order'), ROOT ('ORDERS'), ELEMENTS XSINIL

**Dodawanie własnej specyfikacji namespace** → za pomocą składni WITH XMLNAMESPACES('http://www.adventureWorks.com/report1' as av) przede SELECT. Określasz URI namespace-a, który ma definiować ten dokument - ta strona wcale nie musi isnieć, ale pomaga to rezerwować namespace. Namespace będzie przywoływany przy pomocy prefiksu av. Jeśli w pliku XML autor podał nazwę namespace, to przy otiweraniu namespace również trzeba się powołać na tą nazwę.

Np.

WITH XMLNAMESPACES('http://www.adventureWorks.com/report1' as av)

SELECT

c.CustomerID AS '@av:CustID', c.AccountNumber AS '@av:AccNum'

,h.SalesOrderID, h.OrderDate AS 'av:Details/av:Date', h.Status AS 'av:Details/av:Status', h.PurchaseOrderNumber AS 'av:PurchaseOrderNumber'

,d.ProductID AS 'av:ProductID', d.OrderQty AS 'av:OrderQty', d.UnitPrice AS 'av:UnitPrice'

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML PATH ('Order'), ROOT ('ORDERS'), ELEMENTS XSINIL

**Namespace domyślny** → kiedy wpisywanie za każdym razem prefiksu jest zbyt czasochłonne możesz posłużyć się namespace domyślnym - co można zrobić po prostu nie podając prefiksu dla namespaces. Dodaj DEFAULT w poleceniu WITH.

Np.

WITH XMLNAMESPACES(DEFAULT 'http://www.adventureWorks.com/report1')

SELECT

c.CustomerID AS '@CustID', c.AccountNumber AS '@AccNum'

,h.SalesOrderID, h.OrderDate AS 'Details/Date', h.[Status] AS 'Details/Status', h.PurchaseOrderNumber

,d.ProductID, d.OrderQty, d.UnitPrice

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader h ON h.CustomerID = c.CustomerID

JOIN Sales.SalesOrderDetail d ON d.SalesOrderID = h.SalesOrderID

WHERE c.CustomerID IN (11000, 11001, 29672)

ORDER BY c.CustomerID

FOR XML PATH ('Order'), ROOT ('ORDERS'), ELEMENTS XSINIL

**70. XML - eksport danych z wykorzystaniem FOR XML EXPLICIT**

**FOR XML EXPLICIT** → jedna z najbardziej skomplikowanych metod generowania pliku XML w SQL. Wykorzystaj tą metodę jeśli poprzednie nie udadzą się. Dla każdego z pobieranych rekordów trzeba będzie określić na którym poziomie XML ten rekord ma się znaleźć. W XML można powiedzieć, że elementy mają poziomy i coś zawiera się w czymś innym. Poziom elementu określa się za pomocą polecenia UNION. Muszą powstać w SELECT 2 kolumny: Tag (określa o jaki element chodzi) i Parent (określa hierarchię elementu). Przykładowo 1 as Tag, 0 as Parent oznacza, że jest to pierwsza pozycja. Dodaj aliasy dla kolumn. Dodaj kolejne zapytanie SELECT razem z UNION ALL - doklej w określonym miejscu dane z określonej kolumny. Oznacz te dane jako Tag 2 i Parent 1, gdyż jest niżej w hierarchii. Jeszcze kolejny element będzie miał Tag 3, ale rodzicem będzie nadal element z Tag 1 i wybrane kolumny będą stanowić atrybuty tego poziomu. Podsumowując: w pierwszym SELECT/FROM/WHERE tworzysz pierwszy element-rodzica i on ma przypisane wartości, które wynikają z wybranych w tym SELECT kolumn - tych bez NULL, ale jednocześnie musisz stworzyć (razem z NULL) też aliasy kolumn, które zostaną wykorzystane w elementach-córkach. To co jest w aliasie czyli np. !1! odwołuje się do numeru tagu. W następnych tagach-córkach wymagane jest żeby wykorzystać kolumnę przypisaną do elementu-rodzica. Ostatecznie możesz posortować plik XML za pomocą ORDER BY - bez tego dane pojawią się tak, że najpierw rodzic, potem tagi, ale wszystko po kolei, a nie jedno wynikające z drugiego.

Np.

USE AdventureWorks

SELECT

1 as Tag,

0 as Parent,

SalesOrderID as [OrderHeader!1!SalesOrderID],

OrderDate as [OrderHeader!1!OrderDate],

CustomerID as [OrderHeader!1!CustomerID],

NULL as [SalesPerson!2!SalesPersonID],

NULL as [OrderDetail!3!SalesOrderID],

NULL as [OrderDetail!3!LineTotal],

NULL as [OrderDetail!3!ProductID],

NULL as [OrderDetail!3!OrderQty]

FROM Sales.SalesOrderHeader

WHERE SalesOrderID=43659 or SalesOrderID=43661

UNION ALL

SELECT

2 as Tag,

1 as Parent,

SalesOrderID,

NULL,

NULL,

SalesPersonID,

NULL,

NULL,

NULL,

NULL

FROM Sales.SalesOrderHeader

WHERE SalesOrderID=43659 or SalesOrderID=43661

UNION ALL

SELECT

3 as Tag,

1 as Parent,

SOD.SalesOrderID,

NULL,

NULL,

SalesPersonID,

SOH.SalesOrderID,

LineTotal,

ProductID,

OrderQty

FROM Sales.SalesOrderHeader SOH, Sales.SalesOrderDetail SOD

WHERE SOH.SalesOrderID = SOD.SalesOrderID

AND (SOH.SalesOrderID = 43659 or SOH.SalesOrderID = 43661)

ORDER BY [OrderHeader!1!SalesOrderID], [SalesPerson!2!SalesPersonID], [OrderDetail!3!SalesOrderID], [OrderDetail!3!LineTotal]

FOR XML EXPLICIT;

**Zagnieżdżanie danych** → polega na definiowaniu kolejnych elementów-rodziców. Pierwsze SELECT = rodzic, drugie córka, ale trzecie też córka przy czym nie pierwszego, ale drugiego SELECT.

Np.

USE AdventureWorks

SELECT

1 as Tag,

0 as Parent,

SalesOrderID as [OrderHeader!1!SalesOrderID],

OrderDate as [OrderHeader!1!OrderDate],

CustomerID as [OrderHeader!1!CustomerID],

NULL as [SalesPerson!2!SalesPersonID],

NULL as [OrderDetail!3!SalesOrderID],

NULL as [OrderDetail!3!LineTotal],

NULL as [OrderDetail!3!ProductID],

NULL as [OrderDetail!3!OrderQty]

FROM Sales.SalesOrderHeader

WHERE SalesOrderID=43659 or SalesOrderID=43661

UNION ALL

SELECT

2 as Tag,

1 as Parent,

SalesOrderID,

NULL,

NULL,

SalesPersonID,

NULL,

NULL,

NULL,

NULL

FROM Sales.SalesOrderHeader

WHERE SalesOrderID=43659 or SalesOrderID=43661

UNION ALL

SELECT

3 as Tag,

2 as Parent,

SOD.SalesOrderID,

NULL,

NULL,

SalesPersonID,

SOH.SalesOrderID,

LineTotal,

ProductID,

OrderQty

FROM Sales.SalesOrderHeader SOH, Sales.SalesOrderDetail SOD

WHERE SOH.SalesOrderID = SOD.SalesOrderID

AND (SOH.SalesOrderID = 43659 or SOH.SalesOrderID = 43661)

ORDER BY [OrderHeader!1!SalesOrderID], [SalesPerson!2!SalesPersonID], [OrderDetail!3!SalesOrderID], [OrderDetail!3!LineTotal]

FOR XML EXPLICIT;

**71. XML - jak korzystać z pliku XML?**

**XML, jak korzystać z pliku XML** → dane, które dochodzą do administratora często są w formacie XML, często należy z nich korzystać. Nie wszystkie dane można tak uporządkować i opisać żeby umieścić w relacyjnej bazie danych.

**Różnica między txt, a xml** → można zadeklarować tą samą zmienną jako txt lub xml, przy czym różnicą będzie taka, że stworzenie xml stworzy link/odnośnik do danych, które będą zaprezentowane w wygodny sposób.

Np.

DECLARE @txt NVARCHAR(1000)

SET @txt = '<Model><ModelID>196</ModelID><Name>Metalic frame P348</Name></Model>'

SELECT @txt

DECLARE @xml XML

SET @xml = '<Model><ModelID>196</ModelID><Name>Metalic frame P348</Name></Model>'

SELECT @xml

**Umieszczanie kolejnych obiektów XML w tabeli** → stwórz tabelę, nadaj jej ID (klucz podstawowy) + kolumnę z obiektami XML, po czym umieść je z INSERT INTO.

Np.

CREATE TABLE dbo.Models

(Id INT IDENTITY PRIMARY KEY,

ModelXML XML)

INSERT INTO dbo.Models VALUES('<Model><ModelID>196</ModelID><Name>Metalic frame P348</Name></Model>')

INSERT INTO dbo.Models VALUES('<Model><ModelID>197</ModelID><Name>Metalic frame P647</Name></Model>')

SELECT \* FROM dbo.Models

**Umieszczanie kolejnych obiektów XML w tabeli, ale w jednym linku** → jeśli będziesz wstawiać wiele obiektów za pomocą VALUES to znajdą się one pod jednym linkiem, nazwanym jak pierwszy obiekt, ale będą tam wszystkie rekordy z VALUES.

Np.

INSERT INTO dbo.Models VALUES('<Model><ModelID>100</ModelID><Name>Metalic frame P100</Name></Model>

    <Model><ModelID>101</ModelID><Name>Metalic frame P101</Name></Model>

    <Model><ModelID>102</ModelID><Name>Metalic frame P102</Name></Model>')

SELECT \* FROM dbo.Models

**Deklarowanie zmiennej typu XML jako parametr procedury** → podaj nazwę procedury, deklarujesz Id i podaj parametr XML. W przykładzie procedura będzie wyświetlać Id i XML. Wywołanie procedury poprzez podanie Id procedury i inicjujemy wartość będącą fragmentem wartości typu XML.

Np.

CREATE PROCEDURE dbo.TestXML @Id INT, @xml XML

AS

SELECT @id AS ID, @xml AS XMLData

GO

EXEC dbo.TestXML @id=101, @xml = '<Model><ModelID>196</ModelID><Name>Metalic frame P348</Name></Model>'

GO

**(...)**

**SQL Server - Modyfikacja danych, transakcje i błędy**

**72. Polecenie INSERT - wprowadzenie**

**REFERENCES I CREATE SCHEMA** → za pomocą CREATE SCHEMA tworzysz schemat, a za pomocą REFERENCES wymuszasz, żeby wartość wstawiana z takim odwołaniem musiała znajdować się też w kolumnie odwoływanej tabeli.

Np.

USE AdventureWorks

GO

CREATE SCHEMA Office

GO

CREATE TABLE Office.Equipment

(

Id INT IDENTITY PRIMARY KEY,

[Name] NVARCHAR(50) NOT NULL,

IsDurable BIT,

DurabilityPeriodInMonths INT,

IsPersonAssigned BIT,

PersonID INT REFERENCES Person.Person(BusinessEntityID),

Description NVARCHAR(1000),

InventoryDate DATE DEFAULT GETDATE()

)

GO

**INSERT INTO** → pisz INSERT INTO, nazwę tabeli gdzie wstawiamy rekord (nazwy kolumny - nie podajesz kolumny ID), następnie VALUES określa jakie wartości mają do tabeli być wprowadzone (wartości).

Np.

INSERT INTO Office.Equipment ([Name], IsDurable, DurabilityPeriodInMonths, IsPersonAssigned, PersonID, Description, InventoryDate)

VALUES ('PostIt-Yellow',0,NULL,0,NULL,'size small', GETDATE())

GO

SELECT \* FROM Office.Equipment

**73. Polecenie INSERT - szczegóły składniowe**

**Opuszczenie INTO**→ wstawiając rekord można wyrzucić. Przykład pod spodem.

**Opuszczanie zdefiniowanego GETDATE** → można opuścić wartość GETDATE jeśli została ona zdefiniowana w tworzeniu tabeli. Przykład pod spodem.

**Opuszczanie wartości NULL** → wartości, które mają NULL również zarówno w INSERT jak i VALUES mogą zostać pominięte. Przykład pod spodem.

Np.

USE AdventureWorks

GO

INSERT Office.Equipment (Name, IsDurable, DurabilityPeriodInMonths, IsPersonAssigned, PersonID, Description)

VALUES ('PostIt-Green', 0, NULL, 0, NULL, 'size small')

GO

SELECT \* FROM Office.Equipment

INSERT Office.Equipment (Name, IsDurable, IsPersonAssigned, Description)

VALUES ('PostIt-White', 0, 0, 'size small')

GO

SELECT \* FROM Office.Equipment

**DEFAULT** → jeśli nie wiesz co wpisać, zawsze możesz spróbować wpisać DEFAULT w VALUES - zadziała chociażby w przypadku GETDATE. Przykład poniżej.

**Opuszczanie nazw kolumn w INSERT** → jeśli tylko będziesz wpisywać VALUES w odpowiedniej kolejności jak tworzysz tabelę, to nie musisz wpisywać nazw kolumn w poleceniu INSERT. Przykład poniżej.

INSERT Office.Equipment

VALUES('PostIt-Black', 0, NULL, 0, NULL, 'size-small', DEFAULT)

SELECT \* FROM Office.Equipment

**VALUES** → liczba mnoga tego polecenia oznacza, że wprowadzonych może być więcej niż jeden rekord. Wprowadzanie to powinno odbywać się po przecinku w nawiasach okrągłych.

Np.

INSERT INTO Office.Equipment (Name, IsDurable, DurabilityPeriodInMonths, IsPersonAssigned, PersonID, Description)

VALUES ('Pen-Blue', 0, NULL, 0, NULL, 'size small'),

    ('Pen-Black', 0, NULL, 0, NULL, 'size small')

GO

SELECT \* FROM Office.Equipment

**Id jako wartość wewnętrzna** → zwróć uwagę, że Id powinieneś wykorzystywać i traktować jako coś wewnętrznego i abstrakcyjnego, potrzebnego do łączenia tabel, a nie do analizy czy drukowania jako część raportu. Jeśli chcesz mieć jakąś liczbę porządkową to stwórz taką kolejną kolumnę. Jest tak ponieważ manipulując rekordami pojawiać się będą nieścisłości - jeśli spróbujesz dodać rekord, który zwróci błąd bo chociażby nie odnosi się do istniejącego innego rekordu innej tabeli (w ramach REFERENCE) to Id będzie mu przypisane, po czym okaże się, że to błąd i kolejny poprawny rekord będzie miał już kolejne Id. Tamto poprzednie przepadnie.

INSERT INTO Office.Equipment (Name, IsDurable, DurabilityPeriodInMonths, IsPersonAssigned, PersonID, Description)

VALUES ('Pen-Blue', 1, NULL, 2, 99999, 'size small')

GO

INSERT INTO Office.Equipment (Name, IsDurable, DurabilityPeriodInMonths, IsPersonAssigned, PersonID, Description)

VALUES   ('Pen-Blue', 1, NULL, 2, NULL, 'size small')

SELECT \* FROM Office.Equipment

**(...)**

**74. Polecenie DELETE - wprowadzenie**

**Ćwiczenie z DELETE** → żeby nie zniszczyć bezpowrotnie danych, na których ćwiczę, przekopiuję to co mnie interesuje tymczasowo do bazy tempdb

USE tempdb

GO

SELECT \* INTO Product FROM AdventureWorks.Production.Product

GO

SELECT \* INTO OrderHeader FROM AdventureWorks.Sales.SalesOrderHeader

GO

SELECT \* INTO OrderDetail FROM AdventureWorks.Sales.SalesOrderDetail

**DELETE i dobra praktyka** → aby usunąć dane z tabeli napisz DELETE FROM Tabela WHERE identyfikator = x. Zanim jednak to zrobisz upewnij się, że chodzi o ten rekord o który ci chodzi.

SELECT \* FROM Product WHERE ProductID = 316

GO

DELETE FROM Product WHERE ProductID = 316

GO

**75. Kasowanie części rekordów - DELETE TOP**

**tempdb i kasowanie danych** → baza tempdb działa w ten sposób, że po zamknięciu SQL kasuje dokonywane w niej zmiany, co oznacza, że pracując z dodawanymi do niej tabelami musisz je za każdym razem dodać na nowo.

**DELETE TOP kontrolowane** → usuwanie rekordów masowo należy przeprowadzić w kontrolowany sposób, ponieważ samo polecenie TOP usuwa rekordy losowo tak jak SELECT TOP wyświetla je losowo. Zbuduj zapytanie SELECT TOP dla wyświetlenia tylko klucza podstawowego rekordów spełniających interesujący cię warunek, ale jednocześnie odpowiednio posortowany. Zapytanie to stanie się listą rekordów do usunięcia w momencie jak umieścisz je w podzapytaniu dla klauzuli WHERE polecenia DELETE TOP.

USE tempdb

GO

SELECT \* INTO Product FROM AdventureWorks.Production.Product

GO

SELECT \* INTO OrderHeader FROM AdventureWorks.Sales.SalesOrderHeader

GO

SELECT \* INTO OrderDetail FROM AdventureWorks.Sales.SalesOrderDetail

SELECT TOP(5) ProductID FROM Product WHERE Color IS NULL ORDER BY ProductID

DELETE FROM Product

WHERE

ProductID IN

(SELECT TOP(5) ProductID FROM Product WHERE Color IS NULL ORDER BY ProductID)

**DELETE TOP losowo** → jeśli chcę usuwać rekordy losowo korzystam z DELETE TOP w zwykły sposób.

SELECT \* FROM OrderDetail

DELETE TOP(100) FROM OrderDetail

**(...)**

**76. Polecenie UPDATE - Wprowadzenie**

**Ćwiczenie z UPDATE i tempdb** → podobnie jak w przypadku DELETE, ćwicz z UPDATE na bazie tempdb.

Np.

USE tempdb

GO

SELECT \* INTO Product FROM AdventureWorks.Production.Product

GO

SELECT \* INTO OrderHeader FROM AdventureWorks.Sales.SalesOrderHeader

GO

SELECT \* INTO OrderDetail FROM AdventureWorks.Sales.SalesOrderDetail

**UPDATE** → uzupełnianie/aktualizowanie danych tabeli. Wzór: UPDATE Tabela SET Kolumna = 'Nowa wartość' WHERE Kolumna warunek.

Np.

SELECT \* FROM Product WHERE Color IS NULL

UPDATE Product

    SET Color = 'Unknown'

    WHERE Color IS NULL

GO

**Zmiana wartości wielu kolumn naraz** → dodając po przecinku w nowych liniach kolejne elementy klauzuli SET.

Np.

SELECT \* FROM Product WHERE Color = 'White'

UPDATE Product

    SET

    ListPrice = ListPrice\*1.1,

    SafetyStockLevel += 10,

    StandardCost += 0.10

    WHERE Color = 'White'

GO

**UPDATE TOP** → tak jak w przypadku polecenia TOP DELETE, stwórz polecenie SELECT wybierające TOP kluczy podstawowych dla określonego warunku i posortowane według określonych danych. Wykorzystaj to polecenie w klauzuli WHERE IN.

Np.

SELECT TOP(10) ProductID FROM Product WHERE Color = 'Unknown' ORDER BY ListPrice DESC

Update Product

    SET

    ListPrice = 0.9 \* ListPrice

    WHERE

    ProductId IN

    (SELECT TOP(10) ProductID FROM Product WHERE Color = 'Unknown' ORDER BY ListPrice DESC)

**Różnica między SET, a SELECT przy deklarowaniu wartości** → W SQL dwoma najważniejszymi poleceniami do przypisywania zmiennych są SET i SELECT. Polecenie SET służy do wydajnego przypisywania pojedynczych wartości, natomiast polecenie SELECT służy do pobierania danych i przypisywania wielu wartości. Polecenie SET służy przede wszystkim do przypisywania wartości zmiennym. Jest powszechnie stosowane w procedurach składowanych, skryptach lub partiach w celu przechowywania pojedynczej wartości skalarnej. Z drugiej strony polecenie SELECT służy szerszemu celowi. Pobiera dane z jednej lub większej liczby tabel, widoków lub wyrażeń i nie ogranicza się do przypisywania wartości zmiennym. Jest to wszechstronne polecenie używane do wykonywania zapytań i zwracania zestawów wyników. Użyj polecenia SET, gdy masz do czynienia z wartościami skalarnymi, szczególnie w przypadku przypisań zmiennych w skryptach lub procedurach składowanych. Z drugiej strony polecenie SELECT jest poleceniem do wykonywania zapytań i pobierania danych z tabel lub widoków, oferując bardziej kompleksowe podejście podczas pracy z zestawami wyników.

**UPDATE według obliczonej wartości** → zamiast SET wykorzystaj SELECT.

Np.

USE tempdb

GO

SELECT \* INTO dbo.SalesPerson FROM AdventureWorks.Sales.SalesPerson

GO

DECLARE @averageSalesYTD DECIMAL

SELECT @averageSalesYTD = AVG(SalesYTD) FROM SalesPerson

UPDATE SalesPerson

    SET Bonus += 500

    WHERE SalesYTD > @averageSalesYTD

GO

UPDATE SalesPerson

    SET Bonus += 500

    WHERE SalesYTD > (SELECT AVG(SalesYTD) FROM SalesPerson)

SELECT \* FROM dbo.SalesPerson

**(...)**

**77. Tabele tymczasowe**

**Tabele tymczasowe lokalne** → w SQL tabele tymczasowe zaczynają się od #. Tworzy się je jak zwykłe tabele, np. za pomocą CREATE TABLE. Nieważne w której jesteś bazie danych, tabela tymczasowa powstanie w tempdb, ale możesz się do niej odwoływać po prostu za pomocą #. Umieszczanie danych chociażby przez INSERT zasilany SELECT. Ta tabela nie będzie widoczna w innych sesjach. Tabela tymczasowa istnieje tak długo jak długo istnieje sesja. Mimo tego dobrze jest zakończyć istnieje tabeli po jej użytkowaniu poprzez DROP TABLE.

Np.

USE AdventureWorks

GO

CREATE TABLE #Names

(LastName NVARCHAR(100),

FirstName NVARCHAR(100))

GO

INSERT INTO #Names

SELECT p.LastName, p.FirstName

FROM Person.Person p

SELECT \* FROM #Names

GO

**Tabele tymczasowe globalne** → zamiast jednego # wykorzystaj dwa ##. Ta tabela będzie widoczna w innej sesji, ale nie będzie działać jeśli odłączysz się od sesji źródłowej.

Np.

USE AdventureWorks

GO

CREATE TABLE ##Names

(LastName NVARCHAR(100),

FirstName NVARCHAR(100))

GO

INSERT INTO ##Names

SELECT p.LastName, p.FirstName

FROM Person.Person p

SELECT \* FROM ##Names

GO

**78. Zmienne tabelaryczne**

**Zmienne tabelaryczne** → inaczej zmienne tablicowe. Deklarowane są DECLATE i @, ale określając typ tej zmiennej użyj składni TABLE, a TABLE ma strukturę taką, że w nawiasie piszesz jaka to tabela, jakie ma kolumny. Możesz dodawać rekordy pojedynczo lub kopiować z innej tabeli. Zwróć uwagę, że stawianie instrukcji GO, oddzielając poszczególne fragmenty kodu może spowodować, że te po GO mogą nie działać - a to dlatego, że pierwsze GO wykorzystuje zadeklarowanie zmiennej tabelarycznej.

Np.

USE AdventureWorks

GO

DECLARE @ProdNames TABLE (Name NVARCHAR(100))

INSERT INTO @ProdNames VALUES('product1'),('product2')

SELECT \* FROM @ProdNames

INSERT INTO @ProdNames

SELECT Name FROM Production.Product

SELECT \* FROM @ProdNames

**(...)**

**79. Transakcje - wprowadzenie**

**Transakcja SQL** → transakcja to logiczna jednostka pracy, która wykonuje pojedynczą czynność lub wiele czynności w bazie danych. Transakcje mogą składać się z pojedynczej operacji odczytu, zapisu, usuwania lub aktualizacji lub ich kombinacji. Załóżmy, że gdy chcemy wypłacić pieniądze z bankomatu, aplikacja bankomatu wykona tę operację w trzech krokach. Jako pierwszy krok aplikacja sprawdzi saldo konta, a następnie odliczy pieniądze z konta źródłowego. Wraz z tymi dwoma procesami będzie przechowywać dziennik tej czynności wypłaty pieniędzy.  Główną ideą transakcji jest to, że gdy każde z poleceń zwróci błąd, wszystkie modyfikacje zostaną wycofane, aby zapewnić integralność danych. Z drugiej strony, jeśli wszystkie polecenia zostaną pomyślnie ukończone, modyfikacje danych staną się trwałe w bazie danych. W rezultacie, jeśli doświadczymy jakiejkolwiek przerwy w dostawie prądu lub innych problemów podczas wypłacania pieniędzy z bankomatu, transakcje zagwarantują spójność naszego salda. Najlepszą metodą byłoby wykonanie wszystkich tych kroków za pomocą transakcji, ponieważ cztery główne właściwości transakcji umożliwiają wszystkim operacjom dokładniejsze i bardziej spójne. Wszystkie te właściwości są znane jako ACID (atomowość, spójność, izolacja, trwałość) w systemach baz danych relacyjnych, a pierwsza litera ich nazw jest pierwszą literą.

**Transakcja, BEGIN TRANSACTION i COMMIT** → to kilka poleceń SQL, które wykonają się albo wszystkie albo żadne z nich. Załóżmy, że chcę dodać do tabeli dwa rekordy, a jeśli jeden miałby nie działać to niech nie dodaje się żaden z rekordów. Transakcja zaczyna się od BEGIN TRANSACTION. Potem pisz polecenia. Zazwyczaj transakcja kończy się poleceniem COMMIT. COMMIT mówi, że wszystkie te polecenia mają być zapisane do bazy danych.

Np.

USE AdventureWorks

GO

SELECT \* FROM Production.ScrapReason

GO

BEGIN TRANSACTION

INSERT INTO Production.ScrapReason(Name, ModifiedDate) VALUES ('Bad color', GetDate())

INSERT INTO Production.ScrapReason(Name, ModifiedDate) VALUES ('Bad quality', GetDate())

COMMIT

SELECT \* FROM Production.ScrapReason

GO

**ROLLBACK** → ta operacja ma za zadanie wycofać wszystkie czynności, które były wykonywane od słowa BEGIN TRANSACTION.

Np.

USE AdventureWorks

GO

SELECT \* FROM Production.ScrapReason

GO

BEGIN TRANSACTION

INSERT INTO Production.ScrapReason(Name, ModifiedDate) VALUES ('Wrong color', GetDate())

INSERT INTO Production.ScrapReason(Name, ModifiedDate) VALUES ('Wrong quality', GetDate())

ROLLBACK

SELECT \* FROM Production.ScrapReason

GO

**Usuwanie wszystkich danych z tabeli** → odbywa się poprzez klauzulę DELETE FROM Tabela, bez niczego pomiędzy.

Np.

USE tempdb

GO

SELECT \* INTO Production FROM AdventureWorks.Production.Product

GO

BEGIN TRANSACTION

DELETE FROM Production

SELECT \* FROM Production

ROLLBACK

SELECT \* FROM Production

**80. Zagnieżdżanie transakcji**

**QQTRANCOUNT** → to zmienna, która przechowuje informacje o ilości otwartych w danej chwili transakcji.

Np.

SELECT @@TRANCOUNT

GO

BEGIN TRAN

SELECT @@TRANCOUNT

**Zagnieżdżanie transakcji** → transakcje można zagnieżdżać, można tworzyć transakcje zewnętrzne i wewnętrzne. Można robić to do 32 poziomów. Zamykanie kolejnych transakcji odbywa się normalnie przez COMMIT, ale na odpowiednich poziomach. Inaczej zachowa się ROLLBACK, który cofnie wszystkie transakcje dotychczas wprowadzone.

Np.

SELECT @@TRANCOUNT

GO

BEGIN TRAN

SELECT @@TRANCOUNT

BEGIN TRAN

SELECT @@TRANCOUNT

BEGIN TRAN

SELECT @@TRANCOUNT

COMMIT TRAN

SELECT @@TRANCOUNT

COMMIT TRAN

SELECT @@TRANCOUNT

COMMIT TRAN

SELECT @@TRANCOUNT

BEGIN TRAN

SELECT @@TRANCOUNT

BEGIN TRAN

SELECT @@TRANCOUNT

BEGIN TRAN

SELECT @@TRANCOUNT

ROLLBACK

SELECT @@TRANCOUNT

**Kopiowanie całej tabeli do tabeli tymczasowej** → nie deklarujesz ani nie definiujesz żadnej tabeli, tylko piszesz: SELECT \* INTO  Nazwa\_nowej\_tabeli FROM Baza\_danych.Schemat.Tabela.

Np.

USE tempdb

GO

SELECT \* INTO Production FROM AdventureWorks.Production.Product

SELECT \* FROM Production

BEGIN TRAN

UPDATE Production

SET ListPrice = 0

SELECT @@TRANCOUNT

BEGIN TRAN

UPDATE Production

SET Color = 'Silver'

SELECT @@TRANCOUNT

COMMIT TRAN

SELECT Color, ListPrice FROM Production

SELECT @@TRANCOUNT

ROLLBACK

SELECT Color, ListPrice FROM Production

SELECT @@TRANCOUNT

**81. Transakcja w praktyce**

**SCOPE\_IDENTITY** → Zwraca ostatnią wartość identity wstawioną do kolumny identity w tym samym zakresie.

Np.

USE Tempdb

GO

CREATE TABLE Products

(

 ProductId INT IDENTITY PRIMARY KEY,

 Name NVARCHAR(50) NOT NULL,

 Amount INT,

 Price FLOAT

 )

GO

CREATE TABLE Orders

 (

 OrderId INT IDENTITY PRIMARY KEY,

 Customer NVARCHAR(50) NOT NULL,

 OrderDate DATE

 )

GO

CREATE TABLE OrderDetails

(

 OrderDetailId INT IDENTITY PRIMARY KEY,

 OrderId INT REFERENCES Orders(OrderId) NOT NULL,

 ProductId INT REFERENCES Products(ProductId) NOT NULL,

 Amount INT,

 PRICE FLOAT

)

GO

INSERT Products (Name, Amount, Price)

VALUES ('Barcelona Tourist Guide', 3, 5.5),

  ('Rome Tourist Guide', 1, 3.5),

  ('Lisbon Tourist Guide', 10, 4.4)

GO

DECLARE @CustomerName NVARCHAR(50) = 'Travel Office Colombus'

DECLARE @ProductId INT = 2

DECLARE @AmountToBuy INT = 1

BEGIN TRAN

INSERT INTO Orders(Customer, OrderDate) VALUES (@CustomerName, GETDATE())

DECLARE @OrderId INT = SCOPE\_IDENTITY()

**Praktyczny przykład** → poniżej przykład.

USE Tempdb

GO

CREATE TABLE Products

(

 ProductId INT IDENTITY PRIMARY KEY,

 Name NVARCHAR(50) NOT NULL,

 Amount INT,

 Price FLOAT

 )

GO

CREATE TABLE Orders

 (

 OrderId INT IDENTITY PRIMARY KEY,

 Customer NVARCHAR(50) NOT NULL,

 OrderDate DATE

 )

GO

CREATE TABLE OrderDetails

(

 OrderDetailId INT IDENTITY PRIMARY KEY,

 OrderId INT REFERENCES Orders(OrderId) NOT NULL,

 ProductId INT REFERENCES Products(ProductId) NOT NULL,

 Amount INT,

 PRICE FLOAT

)

GO

INSERT Products (Name, Amount, Price)

VALUES ('Barcelona Tourist Guide', 3, 5.5),

  ('Rome Tourist Guide', 1, 3.5),

  ('Lisbon Tourist Guide', 10, 4.4)

GO

DECLARE @CustomerName NVARCHAR(50) = 'Travel Office Colombus'

DECLARE @ProductId INT = 2

DECLARE @AmountToBuy INT = 1

BEGIN TRAN

INSERT INTO Orders(Customer, OrderDate) VALUES (@CustomerName, GETDATE())

DECLARE @OrderId INT = SCOPE\_IDENTITY()

DECLARE @Price FLOAT

SELECT @Price = Price FROM Products WHERE ProductId = @ProductID

INSERT INTO OrderDetails (OrderId, ProductId, Price, Amount) VALUES (@OrderId, @ProductID, @Price, @AmountToBuy)

UPDATE Products SET Amount = Amount - @AmountToBuy WHERE ProductId = @ProductId

IF (SELECT Amount FROM Products WHERE ProductId = @ProductId) <0

BEGIN

PRINT 'Product not available in this amount. Transaction rolled back!'

ROLLBACK

END

ELSE

BEGIN

PRINT 'Transaction accepted.'

COMMIT

END

**82. Dobre praktyki w pracy z transakcjami i nie tylko**

**Dobre praktyki** → jeśli jesteś w stanie zweryfikować czy transakcja ma szansę się udać przed rozpoczęciem transakcji, to zrób to. Oznacza to, że w poniższym przykładzie najpierw powinieneś sprawdzić za pomocą klauzuli IF czy ilość produktów jest większa od zero. Tak czy inaczej dokonaj transakcji, podczas twojego sprawdzania "na oko", ktoś inny mógł już wprowadzić zmiany do bazy danych.

**SQL Server - Moje notatki**

**83. Views, pisanie widoków**

**Kurs**: https://www.youtube.com/watch?v=nEAAtj8jbBI

**Tworzenie widoków i ich zastosowanie** → widok tworzysz poprzez zwyczajne zapytanie SELECT/FROM poprzedzone instrukcją CREATE VIEW Nazwa\_widoku AS. Powód dla tworzenia widoków może być taki, że twoje zapytanie SELECT może być długie/skomplikowane/narażone na błędy. Nie musisz za każdym razem pisać zapytania SELECT na nowo, możesz odwołać się do widoku poprzez SELECT \* FROM  Widok. Możesz modyfikować początkowe SELECT tak, żeby widok nie zawierał określonych danych. Usuwanie widoku: DROP VIEW IF EXISTS Nazwa\_widoku. Ponieważ działa to jak SELECT możesz wybierać określone kolumny, które mają się wyświetlać. Oczywiście jeśli dane źródłowe się zmienią to należy stworzyć widok na nowo za pomocą zaktualizowanych danych.

Np.

USE AdventureWorks

DROP VIEW IF EXISTS Equipment

GO

CREATE VIEW Equipment AS

SELECT \* FROM Office.Equipment

WHERE Name = 'Pen-Blue'

GO

SELECT \* FROM Equipment

**84. PIVOT**

**Kurs**: <https://www.youtube.com/watch?v=ozy31aJpW-o>

**PIVOT** → to wyrażenie, które pozwala na rotację danych z wierszy do kolumn. Przydatne do raportowania i pomaga w analizie i wizualizacji danych. Aby dokonać rotacji potrzebujesz polecenia SELECT zwracającego dane, które cię interesują - będzie to podzapytanie, za którym znajdzie się polecenie PIVOT, które w argumentach zawierać będzie funkcję agregującą (np. SUM), zagregowaną kolumnę FOR kolumna z której dane znajdą się w nowych nagłówkach po rotacji IN nazwy nagłówków. Wszystko to znajduje się w zapytaniu SELECT, z zaznaczonymi, interesującymi cię nagłówkami. Aliasujesz też podzapytanie i polecenie PIVOT. Kolejność pisania kodu: przygotuj podzapytanie, polecenie PIVOT, polecenie nadrzędne SELECT, aliasowanie.

Np.

USE AdventureWorks

GO

SELECT

ProductID,

ISNULL([January], 0) AS [January],

ISNULL([February], 0) AS [February],

ISNULL([March], 0) AS [March],

ISNULL([April], 0) AS [April],

ISNULL([May], 0) AS [May],

ISNULL([June], 0) AS [June],

ISNULL([July], 0) AS [July],

ISNULL([August], 0) AS [August],

ISNULL([September], 0) AS [September],

ISNULL([October], 0) AS [October],

ISNULL([November], 0) AS [November],

ISNULL([December], 0) AS [December]

FROM

(

SELECT

ProductID,

OrderQty,

DATENAME(MONTH, ModifiedDate) AS [Month]

FROM Sales.SalesOrderDetail

) AS Src

PIVOT

(

SUM(OrderQty)

FOR [Month] IN ([January], [February], [March], [April], [May], [June], [July], [August], [September], [October], [November], [December])

) AS Pvt

**84. Importowanie danych z pliku Excel, txt, CSV**

**Kurs**: <https://www.youtube.com/watch?v=14FpoXKTEJw>

**Kurs**: <https://www.youtube.com/watch?v=JVP9frj2VSQ>

**Fix problemu z importem pliku Excel**: <https://www.youtube.com/watch?v=t7C151yxwcY>

**Importowanie danych z pliku CSV** → w przykładzie tworzysz nową bazę danych i do niej importujesz plik. Klik na węzeł "Databases" i Create New Database. Klik na bazę danych, Tasks, Import, Next, wybierz źródło (Flat File Source), Browse, wybierz plik, Next, wybierz destynację pliku (czyli twój serwer), wybierz sposób prezentacji danych w bazie danych, Next, zobaczysz widok tabeli oraz separatory wierszy/kolumn, Next, Określ bazę danych gdzie umieszczasz dane, Next, Next, Finish.

Np.

USE Names

GO

SELECT \* FROM dbo.names

DROP TABLE dbo.names

**Importowanie z pliku Excel** → dwa sposoby. Pierwszy to po prostu zapisanie pliku Excel (xlsx) jako CSV, po czym przeprowadzenie operacji importu z CSV. Drugi sposób: Tasks, Import Data, Data Source: MS Excel, wybierz wersję Excela, Browse, wybierz plik plik Excel, Destynacja czyli twój serwer, Specify table: Copy data, wybierz arkusze, Finish.

Np.

USE Names

GO

SELECT \* FROM dbo.Sprzedaż

DROP TABLE dbo.Sprzedaż

**Importowanie z pliku txt** → Tasks, Import Data, Data Source: Flat file, wybierz plik, określ jak identyfikować dane, Destynacja czyli twój serwer, Specify table: Copy data, Finish.

Np.

USE Names

GO

SELECT \* FROM Names.dbo.data

DROP TABLE dbo.data

**85. ALTER**

**ALTER TABLE** → Instrukcja ALTER TABLE służy do dodawania, usuwania lub modyfikowania kolumn w istniejącej tabeli. Instrukcja ALTER TABLE służy również do dodawania i usuwania różnych ograniczeń w istniejącej tabeli.

**ADD** → funkcja dodająca kolumnę w tabeli. Przed ADD nazwa\_kolumny typ\_danych musisz dodać instrukcję ALTER TABLE nazwa\_tabeli.

Np.

USE Names

GO

SELECT \* FROM dbo.names

GO

ALTER TABLE dbo.names

ADD full\_name NVARCHAR(50)

**DROP COLUMN** → usuwanie kolumny. Przed DROP COLUMN nazwa\_kolumny musisz dodać instrukcję ALTER TABLE nazwa\_tabeli.

Np.

ALTER TABLE dbo.names

DROP COLUMN full\_name

**Zmiana nazwy kolumny** → inaczej niż w przypadku poprzednich funkcji nie trzeba używać ALTER TABLE. Zastosuj wzór: EXEC sp\_rename 'tabela\_kolumna', 'nowa\_nazwa', 'COLUMN'

Np.

ALTER TABLE dbo.names

ADD full\_name NVARCHAR(50)

EXEC sp\_rename 'dbo.names.full\_name', 'full\_name\_new', 'COLUMN'

SELECT \* FROM dbo.names

**ALTER COLUMN** → zmienia typ danych w kolumnie. Wzór: ALTER TABLE nazwa\_tabeli ALTER COLUMN nazwa\_kolumny nowy\_typ\_zmiennych.

Np.

ALTER TABLE dbo.names

ALTER COLUMN full\_name\_new INT

SELECT \* FROM dbo.names

**86. Back-up and restore**

**Kurs**: <https://www.youtube.com/watch?v=mr8Lpkx5yag>

**Tworzenie kopii zapasowej** → prawy na bazę danych, Tasks, wybierz typ kopii (pełna, wybrane dane, logi), wybierz destynację kopii (usuń starą, dodaj nową + stwórz nazwę kopii), Ok/stwórz skrypt operacji.

Np.

BACKUP DATABASE [Names] TO  DISK = N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Backup\Names.bak' WITH

NOFORMAT,

NOINIT,

NAME = N'Names-Full Database Backup',

SKIP,

NOREWIND,

NOUNLOAD,

STATS = 10

GO

**Wczytywanie kopii zapasowej** → prawy na węzeł Databases, Restore database, Device, wielokropek, Add, znajdź kopię zapasową, ok, ok, nadaj nazwę bazie danych, przejdź do strony File, sprawdź zgodność pliku mdf i ldf, przejdź do strony Options, wyłącz Take tail-log backup before restoring state, potwierdź wszystko Ok/utwórz skrypt.

Np.

USE [master]

RESTORE DATABASE [Names\_back-up] FROM  DISK = N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Backup\Names.bak' WITH

FILE = 1,

MOVE N'Names' TO N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\Names\_back-up.mdf',

MOVE N'Names\_log' TO N'C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\Names\_back-up\_log.ldf',

NOUNLOAD,

STATS = 5

GO

**87. Index**

**Kurs**: <https://www.youtube.com/watch?v=YuRO9-rOgv4>

**Indeksy w SQL** → są różne typy indeksów, ale dwa podstawowe to: Clustered i Non-Clustered czyli zgrupowane i nie-zgrupowane. Przykładem indeksu zgrupowanego jest Id, klucz podstawowy. Możesz podejrzeć w jaki sposób SQL poszukuje danych, które wyciągasz i potwierdzić określony sposób użytkowania. Zaznacz "Include Actual Execution Plan" albo Crtl+M, uruchom zapytanie, najedź na operację, która będzie podpisana właśnie jako sposób indeksowania, w Operation będziesz mieć tą informację jeszcze raz. Zwróć też uwagę na "Number of Rows Read", co informuje cię ile wierszy musiał przeszukać, jaka była ścieżka. Indeks zatem jest narzędziem pozwalającym zapytaniu znaleźć dane.

Np.

USE AdventureWorks

GO

SELECT \* FROM Person.BusinessEntity WHERE BusinessEntityID = 1000

GO

**Indeks scan** → jeśli nie ma indeksu pomagającego wyszukać dane w zapytaniu, SQL przejrzy całą kolumnę, co jest mało skuteczne, nawet jeśli działa. Zwróć iwagę na "Number of rows read" czy jest wysoka, tak jak liczba przeszukiwanych wierszy.

Np.

USE AdventureWorks

GO

SELECT \* FROM Production.Product WHERE Name = 'Blade'

GO

**NON CLUSTERED INDEX** → jeśli sam zauważę, lub SQL sam zaproponuje mogę dodać indeks, dzięki któremu SQL nie będzie musiał dokonywać skanu wszystkich wierszy, ale od razu przejść do poprawnego. Można to zrobić automatycznie: Execution Plan, prawym na komunikat "Missing Index", Missing Index Details i uruchom kod, możesz też dodać nazwę indeksu.

Np.

USE AdventureWorks

GO

CREATE NONCLUSTERED INDEX Product\_Name

ON AdventureWorks.Production.Product ([Name])

**88. RANK, DENSE\_RANK i ROW\_NUMBER**

**Kurs**: <https://www.youtube.com/watch?v=v8C3xEvlwgE>

**RANK, DENSE\_RANK i ROW\_NUMBER** → każda z tych funkcji służy nadawania rang określonym wartościom - różnica między nimi polega na tym jak SQL zachowa się w przypadku remisów, czyli sytuacji gdzie są dwa lub więcej rekordy o tej samej wartości.

**ROW\_NUMBER** → funkcja nadaje numerację od 1 dla każdego wiersza, po prostu numeruje wiersze i numery umieszcza w nowo powstałej kolumnie. Każdy wiersz otrzymuje unikalny numer, nie bierze pod uwagę remisów. Wzór: SELECT kolumna\_1, kolumna\_2, ROW\_NUMBER() OVER (ORDER BY [kolumna\_sortująca] AS [row\_number]

Np.

SELECT [Name],

  [Score],

  ROW\_NUMBER() OVER (ORDER BY [Score]) AS [row\_number]

FROM   (VALUES ('A', 100),

  ('B', 101),

  ('C', 101),

  ('D', 101),

  ('E', 102),

  ('F', 103)) AS Temp([Name],[Score])

ORDER BY [Score] ASC

**RANK** → funkcja nadaje numerację od 1 dla każdego wiersza, po prostu numeruje wiersze i numery umieszcza w nowo powstałej kolumnie, ale tym razem nadaje rangę na podstawie wybranej kolumny i jeśli pojawi się remis to każdy element remisujący otrzyma tą samą rangę. Liczby, które znajdowałyby się tu zostaną pominięte i kolejna liczba nie remisowa nie będzie następującą niż to co wstawiono w remisie. Wzór: SELECT kolumna\_1, kolumna\_2, RANK() OVER (ORDER BY [kolumna\_sortująca] AS [rank].

Np.

SELECT [Name],

  [Score],

  RANK() OVER (ORDER BY [Score]) AS [row\_number]

FROM   (VALUES ('A', 100),

  ('B', 101),

  ('C', 101),

  ('D', 101),

  ('E', 102),

  ('F', 103)) AS Temp([Name],[Score])

ORDER BY [Score] ASC

**DENSE\_RANK** → funkcja nadaje numerację od 1 dla każdego wiersza, po prostu numeruje wiersze i numery umieszcza w nowo powstałej kolumnie, ale tym razem nadaje rangę na podstawie wybranej kolumny i jeśli pojawi się remis to każdy element remisujący otrzyma tą samą rangę. Kolejne liczby nie-remisowe będą kontynuować na zasadzie +1. Wzór: SELECT kolumna\_1, kolumna\_2, DENSE\_RANK() OVER (ORDER BY [kolumna\_sortująca] AS [dense\_rank].

Np.

SELECT [Name],

  [Score],

  DENSE\_RANK() OVER (ORDER BY [Score]) AS [dense\_rank]

FROM   (VALUES ('A', 100),

  ('B', 101),

  ('C', 101),

  ('D', 101),

  ('E', 102),

  ('F', 103)) AS Temp([Name],[Score])

ORDER BY [Score] ASC

**89. NTILE**

**Kurs**: <https://www.youtube.com/watch?v=AbiGF3v9bHU>

**NTILE** → Rozdziela wiersze w uporządkowanej partycji na określoną liczbę grup. Grupy są numerowane, zaczynając od jednego. Dla każdego wiersza NTILE zwraca numer grupy, do której należy wiersz. Wzór: NTILE (liczba\_grup\_które\_chcę) OVER (ORDER BY Wartość\_sortująca). Jeśli wynik będzie nieparzysty, do pierwszej utworzonej grupy przypisany będzie o jeden wiersz więcej.

Np.\_1

USE tempdb

GO

CREATE TABLE NTILESample (StoreID INT NOT NULL);

INSERT INTO NTILESample VALUES

(1),

(2),

(3),

(4),

(5),

(6),

(7),

(8),

(9),

(10);

SELECT \* FROM NTILESample

SELECT

StoreID,

StoreBuckets = NTILE(2) OVER (ORDER BY StoreID)

FROM NTILESample

Np.\_2

USE AdventureWorks

GO

SELECT \*,

Order\_group = NTILE(10) OVER (ORDER BY SalesOrderID)

FROM Sales.SalesOrderDetail

**90. LAG & LEAD**

**Kurs**: https://www.youtube.com/watch?v=j2u52RQ0qlw

**LAG** → uzyskuje dostęp do danych z poprzedniego wiersza w tym samym zestawie wyników. Z tego względu pierwszym wynikiem będzie NULL bo nie ma skąd brać danych. Wzór: LAG(kolumna\_źródłowa) OVER (ORDER BY kolumna\_sortująca). Obok kolumny źródłowej mogę dodać argument liczbowy dodatkowy, który przesunie wyciąganie wartości o określoną liczbę wierszy.

Np.

USE AdventureWorks

GO

SELECT

PurchaseOrderID,

ProductID,

Day\_of\_year\_DueDate = DATEPART(dy, DueDate),

Previous\_Day\_of\_year\_DueDate = LAG(DATEPART(dy, DueDate)) OVER (ORDER BY PurchaseOrderID),

Two\_days\_before\_from\_Day\_of\_year\_DueDate = LAG(DATEPART(dy, DueDate),2) OVER (ORDER BY PurchaseOrderID)

FROM Purchasing.PurchaseOrderDetail

**LEAD** → uzyskuje dostęp do danych z następnego wiersza w tym samym zestawie wyników. Z tego względu ostatnim wynikiem będzie NULL bo nie ma skąd brać danych. Wzór: LEAD(kolumna\_źródłowa) OVER (ORDER BY kolumna\_sortująca). Obok kolumny źródłowej mogę dodać argument liczbowy dodatkowy, który przesunie wyciąganie wartości o określoną liczbę wierszy.

Np.

USE AdventureWorks

GO

SELECT

PurchaseOrderID,

ProductID,

Day\_of\_year\_DueDate = DATEPART(dy, DueDate),

Next\_Day\_of\_year\_DueDate = LEAD(DATEPART(dy, DueDate)) OVER (ORDER BY PurchaseOrderID),

Two\_days\_after\_from\_Day\_of\_year\_DueDate = LEAD(DATEPART(dy, DueDate),2) OVER (ORDER BY PurchaseOrderID)

FROM Purchasing.PurchaseOrderDetail

**91. PARTITION BY**

**Kurs**: <https://www.youtube.com/watch?v=D6XNlTfglW4>

**PARTITION BY** → często mylone z GROUP BY, które działa inaczej i redukuje liczbę wierszy według sum, średnich i tak dalej. PARTITION BY nie redukuje liczby wierszy. Wzór: Funkcja\_agregująca(kolumna\_grupująca) OVER (PARTITION BY kolumna\_grupująca) AS 'Nazwa\_nowej\_kolumny'. Używamy polecenia SQL PARTITION BY, aby podzielić zbiór wyników na partycje i wykonać obliczenia na każdym podzbiorze partycjonowanych danych. Możemy użyć klauzuli SQL PARTITION BY razem z klauzulą ​​OVER, aby określić kolumnę, na której chcemy wykonać agregację.

Np.

USE AdventureWorks

GO

SELECT \* FROM HumanResources.Employee

SELECT \* FROM Person.Person

SELECT

pp.FirstName,

pp.LastName,

hre.Gender,

DATEDIFF(yyyy, hre.BirthDate, GETDATE()) AS 'Age',

COUNT(hre.Gender) OVER (PARTITION BY hre.Gender) AS 'Total Gender'

FROM HumanResources.Employee hre

JOIN Person.Person pp ON hre.BusinessEntityID = pp.BusinessEntityID

**92. Różnica między systemami zarządzania bazami danych**

**Najpopularniejsze systemy zarządzania bazami danych** → MySQL Postgresql, Microsoft SQL Server.

**MySQL** → działa na wszsytkim, używa standardowego SQL.

**SQL Server** → działa na Linux albo Windows, używa T-SQL.

**PostgreSQL** → działa na wszsytkim, używa standardowego SQL.

**Szybkie sprawdzanie jakie w SQL Server znajdują się tabele i kolumny** → jak na przykładzie.

Np.

SELECT \* FROM sys.TABLES

SELECT \* FROM sys.columns

ORDER BY name

**93. Notatki z Alex the Analyst**

**Sortowanie poziomowe** → jak w Excelu, możesz sortować według kolejności, najpierw po kolumnie x, potem y. Za pomocą ORDER BY i podaj numery kolumn, w odpowiedniej kolejności.

Np.

SELECT \* FROM [Portfolio\_SQL Data Exploration].dbo.Covid\_Deaths

ORDER BY 3,4

Wybierz dane, które będą ci potrzebne.

**Błąd “Divide by zero error encountered”** → zastosuj NULLIF dla elementów dzielenia, które mogą mieć zero, NULLIF(element, 0).

Np.

SELECT location, date, total\_cases, total\_deaths, (NULLIF(total\_deaths,0)/NULLIF(total\_cases, 0))

FROM [Portfolio\_SQL Data Exploration].dbo.Covid\_Deaths

ORDER BY 1,2

**Błędy w sortowaniu liczb/operacji na liczbach** → często wynikają z tego, że dane źródłowe przedstawiają je jako inny typ danych, chociażby nvarchar. Należy wtedy je skonwertować.

**Drill-down effect** → efekt zmniejszania/zwiększania skali. Np. Od kontynentu po kraj, województwo, etc.

**!=** → podobnie jak w Python, w SQL możesz używać symbolu inne niż.

Np.

SELECT date, SUM(new\_cases) AS SumOfCases, SUM(new\_deaths) AS SumOfDeaths, (SUM(NULLIF(new\_deaths,0))/SUM(NULLIF(new\_cases,0))\*100) AS PercentageOfDeaths

FROM [Portfolio\_SQL Data Exploration].dbo.Covid\_Deaths

WHERE continent IS NOT NULL AND new\_cases != 0

GROUP BY date

ORDER BY date ASC

**Błąd “ORDER BY list of RANGE window frame has total size of x bytes. Largest size supported is 900 bytes”** → błąd ten może wyskoczyć kiedy operujesz na zbyt dużych liczbach. W takim przypadku zamiast int skorzystaj z big int, a ponadto dodaj klauzulę ROWS UNBOUNDED PRECEDING, która określa, że ramka zaczyna się od pierwszego wiersza w oknie aż do bieżącego wiersza.

Np.

SELECT

dea.continent,

dea.location,

dea.date,

dea.population,

vac.new\_vaccinations,

SUM(CONVERT(bigint, vac.new\_vaccinations)) OVER (PARTITION BY dea.location ORDER BY dea.location, dea.date ROWS UNBOUNDED PRECEDING) AS vaccinations\_so\_far

FROM Covid\_Deaths AS dea

JOIN Covid\_Vaccinations AS vac

ON dea.location = vac.location AND dea.date = vac.date

WHERE dea.continent IS NOT NULL AND new\_vaccinations != 0

ORDER BY 2,3

**CTE (Common Table Expression)** → CTE, czyli wspólne wyrażenia tablicowe. Upraszczają i poprawiają przejrzystość kodu SQL. W tym zakresie, ich stosowanie nie ma wpływu na wydajność zapytań, tylko na jego czytelność. Oprócz funkcji czysto estetycznej, posiadają jeszcze jedną, specjalną właściwość – ich struktura pozwala na realizację rekurencji. Ich praktycznym zastosowaniem jest chociażby wykorzystanie dla dalszych obliczeń stworzonej kolumny. W poniższym przykładzie w klauzuli SELECT tworzona jest kolumna “vaccinations\_so\_far”, która nie może być wykorzystana do dalszych obliczeń. Żeby ją wykorzystać należy użyć CTE, czyli umieścić zapytanie SELECT w podzapytaniu dla klauzuli WITH tabela\_tymczasowa (ilość\_kolumn\_zgodna\_z\_podzapytaniem) AS podzapytanie SELECT \*, obliczenie\_które\_cię\_interesowało FROM tymczasowa\_tabela.

Np.

WITH PopVsVac (Continent, Location, Date, Population, New\_Vaccinations, vaccinations\_so\_far) AS

(SELECT

dea.continent,

dea.location,

dea.date,

dea.population,

vac.new\_vaccinations,

SUM(CONVERT(bigint, vac.new\_vaccinations)) OVER (PARTITION BY dea.location ORDER BY dea.location, dea.date ROWS UNBOUNDED PRECEDING) AS vaccinations\_so\_far

FROM Covid\_Deaths AS dea

JOIN Covid\_Vaccinations AS vac

ON dea.location = vac.location AND dea.date = vac.date

WHERE dea.continent IS NOT NULL AND new\_vaccinations != 0)

SELECT \*,

(vaccinations\_so\_far/Population)\*100 AS PopulationVsVaccinations

FROM PopVsVac

**Widoki (Views)** → Widok to wirtualna tabela, której zawartość jest definiowana przez zapytanie. Podobnie jak tabela, widok składa się z zestawu nazwanych kolumn i wierszy danych. Jeśli nie jest indeksowany, widok nie istnieje jako zapisany zestaw wartości danych w bazie danych. Wiersze i kolumny danych pochodzą z tabel, do których odwołuje się zapytanie definiujące widok, i są generowane dynamicznie, gdy następuje odwołanie do widoku. Widok działa jako filtr w tabelach bazowych, do których odwołuje się widok. Zapytanie definiujące widok może pochodzić z jednej lub więcej tabel lub z innych widoków w bieżącej lub innych bazach danych. Zapytania rozproszone mogą być również używane do definiowania widoków, które wykorzystują dane z wielu heterogenicznych źródeł. Jest to przydatne na przykład, jeśli chcesz połączyć podobnie ustrukturyzowane dane z różnych serwerów, z których każdy przechowuje dane dla innego regionu Twojej organizacji. Widoki są zazwyczaj używane do skupiania, upraszczania i dostosowywania postrzegania bazy danych przez każdego użytkownika. Widoki mogą być używane jako mechanizmy bezpieczeństwa, umożliwiając użytkownikom dostęp do danych za pośrednictwem widoku, bez udzielania użytkownikom uprawnień do bezpośredniego dostępu do tabel bazowych zapytania. Widoki mogą być używane do zapewnienia wstecznie kompatybilnego interfejsu do emulowania tabeli, która kiedyś istniała, ale której schemat uległ zmianie. Widoki mogą być również używane podczas kopiowania danych do i z programu SQL Server w celu poprawy wydajności i partycjonowania danych. Wzór tworzenia widoków: CREATE VIEW nazwa\_widoku AS SELECT kolumna1, kolumna2, ... kolumnaN FROM nazwa\_tabeli WHERE (warunki). Po stworzeniu i odpaleniu widoku (SELECT TOP x rows) żeby pozbyć się czerwonych symboli błędów użyj Crtl+Shift+R, które odświeży widok. Widoku można też teraz użyć jako tabeli przy SELECT.

Np.

Create View PercentPopulationVaccinated AS

SELECT

dea.continent,

dea.location,

dea.date,

dea.population,

vac.new\_vaccinations,

SUM(CONVERT(bigint, vac.new\_vaccinations)) OVER (PARTITION BY dea.location ORDER BY dea.location, dea.date ROWS UNBOUNDED PRECEDING) AS vaccinations\_so\_far

FROM Covid\_Deaths AS dea

JOIN Covid\_Vaccinations AS vac

ON dea.location = vac.location AND dea.date = vac.date

WHERE dea.continent IS NOT NULL AND new\_vaccinations != 0

**PARSENAME** → służy do wyodrębniania określonych części nazwy obiektu, takich jak nazwa tabeli, nazwa kolumny lub nazwa bazy danych, na podstawie określonego ogranicznika. Jest to szczególnie przydatne, gdy masz do czynienia z obiektami, które mają wiele części oddzielonych ogranicznikiem, takim jak kropka („.”), i musisz pobrać określoną część. Wzór: obiekt dzielony, element, który wyciągasz. Zwróć uwagę, że PARSENAME wyciąga rzeczy od tyłu, czyli żeby wyciągnąć pierwszy element musisz podać ostatni w kolejności, czyli ostatni ogranicznik.

Np.

SELECT

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 3),

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 2),

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 1)

FROM [Portfolio\_SQL Data Cleaning].dbo.NashvilleHousing

**OVER** → klauzula OVER określa partycjonowanie i kolejność zestawu wierszy przed zastosowaniem powiązanej funkcji okna. Oznacza to, że klauzula OVER definiuje okno lub określony przez użytkownika zestaw wierszy w zestawie wyników zapytania. Następnie funkcja okna oblicza wartość dla każdego wiersza w oknie. Klauzuli OVER można używać z funkcjami w celu obliczenia wartości zagregowanych, takich jak średnie kroczące, agregaty skumulowane, sumy bieżące lub wyniki N najlepszych na grupę.

**Znajdowanie duplikatów** → żeby znaleźć duplikaty skorzystaj z połączenia ROW\_NUMBER, OVER, PARTITION BY, ORDER BY. Patrząc od wewnątrz zapytania do PARTITION BY wybierasz kolumny, które będą różnicować i odznaczać występowanie potencjalnych duplikatów - czyli takie, które z założenia powinny być unikalne dla każdego wiersza.ORDER BY będzie na postawie jakiejś unikalnej wartości. PARTITION BY dzieli podany zestaw zbiorów na partycje. I teraz te partycje zostaną ponumerowane za pomocą ROW\_NUMBER() OVER, czyli dla każdej partycji zacznie się numerowanie wierszy od nowa, a że w założeniu każda lub prawie każda partycja jest unikalna to będą same 1 albo tylko niewiele 2, 3, itd. - i to są twoje duplikaty.

Np.

SELECT \*,

    ROW\_NUMBER() OVER (

    PARTITION BY ParcelID,

      PropertyAddress,

      SalePrice,

      SaleDate,

      LegalReference

      ORDER BY

    UniqueID

    ) row\_num

FROM [Portfolio\_SQL Data Cleaning].dbo.NashvilleHousing

**Usuwanie duplikatów** → umieść znalezione duplikaty w CTE i wyróżnij je za pomocą klauzuli WHERE > 1, bo interesują nas duplikaty. Najpierw użyj SELECT żeby je znaleźć po czym zamień SELECT na DELETE żeby usunąć.

Np.

WITH RowNumCTE AS

(

SELECT \*,

ROW\_NUMBER() OVER (

PARTITION BY ParcelID,

PropertyAddress,

SalePrice,

SaleDate,

LegalReference

ORDER BY

UniqueID

) row\_num

FROM [Portfolio\_SQL Data Cleaning].dbo.NashvilleHousing

)

--SELECT \*

DELETE

FROM RowNumCTE

WHERE row\_num > 1

--ORDER BY PropertyAddress

**94. Notatki z portfolio**

UPDATE dbo.Dochody\_ogółem\_Klasyfikacja$

   SET Income\_total\_classification = NULL

 WHERE Income\_total\_classification = '-' or Income\_total\_classification = '0,00'

GO

SELECT \* FROM Dochody\_ogółem\_Klasyfikacja$

UPDATE dbo.Wydatki\_ogółem\_Klasyfikacja$

   SET Expenditure\_total\_classification = NULL

 WHERE Expenditure\_total\_classification = 0

GO

SELECT \* FROM Wydatki\_ogółem\_Klasyfikacja$

**Zmiana typów danych** → za pomocą UPDATE i REPLACE. Przykład poniżej.

Np.

ALTER TABLE Dochody\_ogółem\_Klasyfikacja$

ALTER COLUMN Code int;

ALTER TABLE Dochody\_ogółem\_Klasyfikacja$

ALTER COLUMN [Year] int;

UPDATE Dochody\_ogółem\_Klasyfikacja$

SET Income\_total\_classification = REPLACE(Income\_total\_classification,',','.') FROM Dochody\_ogółem\_Klasyfikacja$

ALTER TABLE Dochody\_ogółem\_Klasyfikacja$

ALTER COLUMN Income\_total\_classification real;

SELECT \* FROM Dochody\_ogółem\_Klasyfikacja$

SELECT Income\_total\_classification FROM Dochody\_ogółem\_Klasyfikacja$ WHERE Income\_total\_classification = 14104017367.88

**Tworzenie klucza podstawowego** → przejdź do SQL Server, Narzędzia, Opcje, Projektanci, Projektanci tabel i baz danych i odznacz Zapobiegaj zapisywaniu zmian, które wymagają ponownego utworzenia tabeli. Otwórz widok projektu tabeli i przewiń kolumnę w górę i w dół i zapisz zmiany. Przywróć potem ustawienia.

Np.

ALTER TABLE Dochody\_budżetów\_województw$

ADD ExInID int IDENTITY(1,1)

ALTER TABLE Dochody\_budżetów\_województw$

ADD CONSTRAINT PK\_Dochody\_budżetów\_województw$\_ExInID PRIMARY KEY CLUSTERED (ExInID)

**ALTER TABLE i UPDATE** → należy dodawać GO, żeby wykonać działania osobno.

Np.

ALTER TABLE Dochody\_na\_1\_mieszkańca$

ADD ModifiedDate DATETIME

GO

UPDATE Dochody\_na\_1\_mieszkańca$ SET ModifiedDate = GETDATE() WHERE ModifiedDate IS NULL

GO

**Obliczenia na wielu zadeklarowanych zmiennych** → jeśli chcesz dokonywać obliczeń na wielu zadeklarowanych zmiennych to pamiętaj deklarować je jedna pod drugą.

Np.

DECLARE @PopulationPL2002 bigint

DECLARE @PopulationPL2023 bigint

DECLARE @IncomePL2002 bigint

DECLARE @IncomePL2023 bigint

DECLARE @IncomePersonPL2002 bigint

DECLARE @IncomePersonPL2023 bigint

SET @PopulationPL2002 = 38218531

SET @PopulationPL2023 = 37636508

SET @IncomePL2002 = (SELECT Budget\_income FROM Dochody\_budżetów\_województw$ WHERE Voivodeship = 'POLSKA' AND [Year] = 2002)

SET @IncomePL2023 = (SELECT Budget\_income FROM Dochody\_budżetów\_województw$ WHERE Voivodeship = 'POLSKA' AND [Year] = 2023)

SET @IncomePersonPL2002 = (SELECT Income\_per\_capita FROM Dochody\_na\_1\_mieszkańca$ WHERE Voivodeship = 'POLSKA' AND [Year] = 2002)

SET @IncomePersonPL2023 = (SELECT Income\_per\_capita FROM Dochody\_na\_1\_mieszkańca$ WHERE Voivodeship = 'POLSKA' AND [Year] = 2023)

**ROUND i zasady matematyki** → drugi argument = 0 zapewni zaokrąglanie według zasad matematyki.

Np.

DECLARE @random float

SET @random = RAND()\*10

SELECT @random, ROUND(@random,0)

**Wypełnianie kolumny zawartością innej z CASE oraz IIF** → jak w przykładzie.

Np.

ALTER TABLE Wydatki\_ogółem\_Klasyfikacja$

ADD Voivodeship\_eng NVARCHAR(20)

UPDATE Wydatki\_ogółem\_Klasyfikacja$

SET Voivodeship\_eng =

CASE

     WHEN CONVERT(NVARCHAR(20), Code) = '200000' THEN 'Lower Silesian'

     WHEN CONVERT(NVARCHAR(20), Code) = '400000' THEN 'Kuyavian-Pomeranian'

     WHEN CONVERT(NVARCHAR(20), Code) = '600000' THEN 'Lublin'

     WHEN CONVERT(NVARCHAR(20), Code) = '800000' THEN 'Lubusz'

     WHEN CONVERT(NVARCHAR(20), Code) = '1000000' THEN 'Łódź'

     WHEN CONVERT(NVARCHAR(20), Code) = '1200000' THEN 'Lesser Poland'

     WHEN CONVERT(NVARCHAR(20), Code) = '1400000' THEN 'Masovian'

     WHEN CONVERT(NVARCHAR(20), Code) = '1600000' THEN 'Opole'

     WHEN CONVERT(NVARCHAR(20), Code) = '1800000' THEN 'Subcarpathian'

     WHEN CONVERT(NVARCHAR(20), Code) = '2000000' THEN 'Podlaskie'

     WHEN CONVERT(NVARCHAR(20), Code) = '2200000' THEN 'Pomeranian'

     WHEN CONVERT(NVARCHAR(20), Code) = '2400000' THEN 'Silesian'

     WHEN CONVERT(NVARCHAR(20), Code) = '2600000' THEN 'Holy Cross'

     WHEN CONVERT(NVARCHAR(20), Code) = '2800000' THEN 'Warmian-Masurian'

     WHEN CONVERT(NVARCHAR(20), Code) = '3000000' THEN 'Greater Poland'

     WHEN CONVERT(NVARCHAR(20), Code) = '3200000' THEN 'West Pomeranian'

     ELSE 'Poland'

END

SELECT \* FROM Wydatki\_ogółem\_Klasyfikacja$

ALTER TABLE Wydatki\_ogółem\_Klasyfikacja$

ADD Poland\_region NVARCHAR(20)

UPDATE Wydatki\_ogółem\_Klasyfikacja$

SET Poland\_region =

IIF(Code = 0, 'Poland', IIF(Code = 200000 OR Code = 400000 OR Code = 800000 OR Code = 1600000 OR Code = 2200000 OR Code = 3000000 OR Code = 3200000, 'West\_Poland','East\_Poland'))

SELECT \* FROM Wydatki\_ogółem\_Klasyfikacja$

**Klonowanie tabeli z SELECT INTO** → jak w przykładzie.

Np.

SELECT \* INTO Dochody\_na\_1\_mieszkańca$\_EvenYears FROM Dochody\_na\_1\_mieszkańca$

DELETE FROM Dochody\_na\_1\_mieszkańca$\_EvenYears WHERE [Year] % 2 = 1

Value increment, sumowanie kolejnych rekordów → jak w przykładzie.

Np.

BEGIN TRANSACTION

DELETE FROM Dochody\_budżetów\_województw$ WHERE NOT Voivodeship = 'POLSKA'

SELECT [Year], Voivodeship, SUM(Budget\_income) OVER (ORDER BY exinid) FROM Dochody\_budżetów\_województw$

ROLLBACK

**Kopiowanie/klonowanie tabel za pomocą** SELECT INTO → jak w przykładzie.

Np.

SELECT \*

INTO Wydatki\_ogółem\_Klasyfikacja$\_2002\_to\_2012