

```
# SQL Course - Day 2
Plan :
1. Grupowanie danych
2. Funkcje SQL
3. Widoki
4. Złożone zapytania
5. Zarządzanie bazą danych - Język DDL Data definition Language

## Section 1 : Grupowanie danych
### Lesson 1 : Funkcje agregujące
#### wartość największa i najmniejsza Funkcje MIN() i MAX()
- Funkcje te umożliwiają wyszukiwanie wartości minimalnej i maksymalnej z danych zwróconych w zapytaniu:
-
    SELECT
        MAX(invoice_total) AS highest,
        MIN(invoice_total) AS lowest
    FROM invoices

- można stosować także do Strings & Data
  //...
  MAX(payment_date) AS latest_payment_date

#### Obliczanie średniej AVG()
- obliczenie średniej z wartości uzyskanej w zapytaniu

#### Funkcja COUNT()
- Funkcja ta umożliwia zwrócenie liczby wszystkich uzyskanych z zapytania rekordów.
- COUNT( something ) zlicza tylko rekordy dla których kolumna something <> NULL
- COUNT(*) - zlicza wszystko
- COUNT(DISTINCT client_id) - zliczy niepowtarzalne wystąpienia klienta dla których wystawiono faktury

#### Funkcja SUM()
- Funkcja ta umożliwia zwrócenie sumy konkretnych wartości na podstawie uzyskanych rezultatów:

#### Pełen przykład :
USE sql_invoicing;

SELECT
    MAX(invoice_total) AS highest,
    MIN(invoice_total) AS lowest,
    AVG(invoice_total) AS average,
    SUM(invoice_total) AS total_sum,
    COUNT(*) AS total_records,
    COUNT(DISTINCT client_id) AS invoicing_clients
FROM invoices
WHERE invoice_date > '2019-07-01'

#### Zadanie 1 -----
Stwórz zapytanie w bazie sql_invoicing, które zwróci wynik : data_range, total_sales , total_payments, due_payments - jak na przykładzie

Rozwiązanie :
USE sql_invoicing;

SELECT
    'First half of 2019' AS date_range,
    SUM(invoice_total) AS total_sales,
    SUM(payment_total) AS total_payments,
    SUM(invoice_total - payment_total) AS due_payments
FROM invoices
```

```
        WHERE invoice_date
              BETWEEN '2019-01-01' AND '2019-06-30'
UNION
        SELECT
            'Second half of 2019' AS date_range,
            SUM(invoice_total) AS total_sales,
            SUM(payment_total) AS total_payments,
            SUM(invoice_total - payment_total) AS due_payments
        FROM invoices
        WHERE invoice_date
              BETWEEN '2019-07-01' AND '2019-12-31'
UNION
        SELECT
            'TOTAL' AS date_range,
            SUM(invoice_total) AS total_sales,
            SUM(payment_total) AS total_payments,
            SUM(invoice_total - payment_total) AS due_payments
        FROM invoices
        WHERE invoice_date
              BETWEEN '2019-01-01' AND '2019-12-31'
```

### 2 : Klauzula GROUP BY  
- grupowanie wyników zapytań  
- składnia :  
SELECT kolumna1, kolumna2, ..., kolumnaN  
FROM tabela1, tabela2, ..., tabelaN  
WHERE warunki (opcjonalnie)  
GROUP BY kolumna1, kolumna2, ..., kolumnaN  
ORDER BY kolumna3 (opcjonalnie)

#### Przykład : szukamy sumy faktur dla poszczególnych klientów

```
SELECT
    client_id,
    SUM(invoice_total) AS total_sales
FROM invoices
GROUP BY client_id          -- pogrupowane wg klientów
ORDER BY total_sales DESC   -- uporządkowane malejąco
```

#### Przykład : grupowanie po kilku kolumnach

```
SELECT
    state,
    city,
    SUM(invoice_total) AS total_sales
FROM invoices i
JOIN clients USING (client_id)
GROUP BY state, city
```

#### Zadanie 2 -----

Zbuduj zapytanie które da rezultat jak poniżej :

Rozwiązanie :

zacznij od poniższego zapytania :

```
SELECT
    date,
    SUM(amount) AS total_payments
FROM payments
GROUP BY date
```

następnie :

1. posortuj dane wg daty ORDER BY date
2. połącz payments[] z payment\_methods[]
3. pogrupuj dodatkowo po payment\_method

```
finally :
    SELECT
        p.date,
        pm.name,
        SUM(p.amount) AS total_payments
    FROM payments p
    JOIN payment_methods pm
        ON p.payment_method = pm.payment_method_id
    GROUP BY date, pm.name
    ORDER BY date

### 3 : Klauzula HAVING
- składnia :
    SELECT kolumna1, kolumna2, ..., kolumnaN
    FROM tabela1, tabela2, ..., tabelaN
    WHERE warunki_where
    GROUP BY kolumna1, kolumna2, ..., kolumnaN
    HAVING warunki_having;

#### Przykład : do ponieszego zapytania dodaj warunek filtrujący rezultat tylko dla klientów, którzy posiadają sprzedaż > 500 $
SELECT
    client_id,
    SUM(invoice_total) AS total_sales
FROM invoices
GROUP BY client_id
- nie możemy dodac warunku w klauzuli WHERE ponieważ total_sales wylicza się dopiero w klauzuli GROUP BY
- musimy uzyć klauzuli HAVING filtrującej dane PO klauzuli GROUP BY
// ...
GROUP BY client_id
HAVING total_sales > 500

- warunek w HAVING posiada te same zasady co warunek w WHERE (nawiasy, operatory AND OR etc)
- w warunku mogą wystąpić `TYLKO` te wartości (kolumny LUB agregaty), który występują w klauzuli SELECT
```

#### Zadanie 3 -----  
W bazie sql\_store stwórz zapytanie, które wyszuka wszystkich klientów ze stanu Wirginia, którzy wydali więcej niż 100\$.

```
USE sql_store;

SELECT *
FROM customers
WHERE state = 'VA'

- dodaj do zapytania informacje o płatnościami : orders [] + order_items[]
    SELECT *
    FROM customers c
    JOIN orders o USING (customer_id)
    JOIN order_items oi USING (order_id)
    WHERE state = 'VA'

- zwracaj jedydnie : c.customer_id, c.first_name, c.last_name
- dodaj sumowanie SUM(oi.quantity * ou.unit_price) AS total_sales
- dodaj GROUP BY
    - best practices - dodajemy WSZYSTKIE kolumny z SELECT
- dodaj filtr total_sales > 100$
- wynik końcowy :
```

```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(oi.quantity * oi.unit_price) AS total_sales
FROM customers c
JOIN orders o USING (customer_id)
JOIN order_items oi USING (order_id)
WHERE state = 'VA'
GROUP BY
    c.customer_id,
    c.first_name,
    c.last_name
HAVING total_sales > 100

### 4 : Operator WITH ROLLUP (optional)
- użyjemy ponownie przykładu z lekcji 2 : GROUP BY
- dodajemy operator WITH ROLLUP - w wyniku otrzymamy dodatkow podsumowanie wierszy
powyżej :
// ...
GROUP BY client_id WITH ROLLUP

- jest to właściwość MySQL - operator ROLLUP nie występuje w standardzie SQL ANSI

##### Zadanie 4 -----
USE sql_invoicing;

SELECT
    payment_method,
    SUM(amount) AS total
FROM payments
GROUP BY payment_method

- dodaj operator ROLLUP
- połącz z tablicą payment_methods[]
- w klauzuli SELECT zamień payment_method > pm.name AS payment_method

- finally :
SELECT
    pm.name AS payment_method,
    SUM(amount) AS total
FROM payments p
JOIN payment_methods pm
ON p.payment_method = pm.payment_method_id
GROUP BY pm.name WITH ROLLUP -- not a "payment_method" which is a label now

-----
## Section 2 : Funkcje wbudowane SQL
### Lesson 1 : Funkcje numeryczne
- funkcja zaokrąglenia ROUND
- SELECT ROUND(5.73)      -- = 6
- SELECT ROUND(5.73, 1)    -- = 5.7

##### Zadanie 1 -----
1. Sprawdź działanie funkcji : TRUNCATE, CEILING(), FLOOR(), ABS(), RAND()

2. zobacz resztę funkcji wpisując w google : mysql numeric functions

### 2 : Funkcje na łańcuchach znakowych (Strings)
- przykłady uzytecznych funkcji
SELECT LENGTH('blue')      -- = 4
SELECT UPPER('Blue')
```

```
SELECT LOWER('BLUe')

SELECT LTRIM('    sky')      -- left trim = 'sky'
SELECT LTRIM('sky    ')     -- right trim = 'sky'
-- SELECT TRIM ('    sky    ') -- = 'sky' - nie ma takiej funkcji

SELECT LEFT('Kindergarten', 4) -- = 'Kind'
SELECT RIGHT('Kindergarten',6) -- = 'garten'

SELECT SUBSTRING('Kindergarten', 3, 5) -- 'nderg'
SELECT SUBSTRING('Kindergarten', 3) -- 'ndergarten'

SELECT LOCATE('N','Kindergarten') -- = 3, not case sensitive
SELECT LOCATE('garte','Kindergarten') -- = 7

SELECT REPLACE('Kindergarten','garten','garden') -- 'Kindergarten'
SELECT CONCAT('first', ' ', 'last') -- 'first last'

USE sql_store;

SELECT CONCAT(first_name, ' ', last_name)
FROM customers

### 3 : Funkcje daty i czasu (DATE and DATETIME)
- przykłady użytecznych funkcji
SELECT NOW(), CURDATE(), CURTIME()

SELECT YEAR(NOW()) -- 2021
- podobnie : MONTH(), DAY(), HOUR(), ETC

SELECT DAYNAME(NOW()) -- Monday (założymy)
SELECT MONTHNAME(NOW())

- w standardzie ANSI mamy inną funkcję
SELECT EXTRACT(YEAR FROM NOW()) -- 2021
```

##### Zadanie 2 -----  
Popraw poniże zapytanie tak aby wskazywało na zamówienia złożone w bieżącym roku

```
SELECT *
FROM orders
WHERE order_date >= '2019-01-01'
```

Rozwiązanie :  
// ...  
WHERE YEAR(order\_date) = YEAR(NOW()) - 2

### 4 : Formatowanie daty i czasu  
- domyślnym formatem w MySQL jest 'YYYY-MM-DD' np. '2021-01-31'

##### Formatowanie daty  
SELECT DATE\_FORMAT(CURDATE(), '%m/%d/%Y') today; -- np 'March 21 2021'

- wypróbuj inne kombinacje jak np %m, %D, %y  
SELECT DATE\_FORMAT(NOW(), "%M %d %Y") today;

- sprawdź możliwości na google : mysql date format string

##### Formatowanie czasu  
SELECT TIME\_FORMAT(NOW(), '%H') 'current hour'

### 5 : Obliczanie daty i czasu  
- przykłady użytecznych funkcji

```
SELECT DATE_ADD(NOW(), INTERVAL 1 DAY)
SELECT DATE_ADD( NOW(), INTERVAL 1 YEAR) 'now plus one year'
```

```
SELECT DATE_SUB( NOW(), INTERVAL 1 YEAR)
SELECT DATEDIFF('2021-01-05', NOW()) -- różnica w dniach
```

- DATEDIFF nie pokazuje różnicę w czasie, nawet jeśli wpiszemy czas do daty np. '2021-01-15 8:52'
- dokładną różnicę w czasie możemy wyliczyć w następujący sposób :  
SELECT TIME\_TO\_SEC('09:02') - TIME\_TO\_SEC('09:00')

### 6 : Funkcje IFNULL oraz COALESCE  
USE sql\_store;

```
SELECT
    order_id,
    IFNULL(shipper_id, 'Not assigned') -- IF shipper_id = NULL THEN put('Not
assigned')
FROM orders
```

```
SELECT
    order_id,
    COALESCE(shipper_id, comments, 'Not assigned' ) AS 'comment from shipper' -- operator trójargumentowy ? :
FROM orders
```

#### Zadanie 3 -----

W bazie sql\_invoicing stwórz zapytanie, które w przypadku braku numeru telefonu, wstawi napis "Unknown".

Rozwiązanie :

```
SELECT
    CONCAT(first_name, ' ', last_name) AS customer,
    IFNULL(phone, 'Unknown') AS phone
FROM customers
```

### 7 : Funkcja IF

- składnia IF(expression, first\_if\_true, second)
- przykład :

```
SELECT
    order_id,
    order_date,
    IF( YEAR(order_date) = YEAR(NOW() ) -2, 'Active', 'Archive') AS category
FROM orders
```

#### Zadanie 4 -----

```
SELECT
    product_id,
    name,
    COUNT(*) AS orders,
    IF( COUNT(*) > 1, 'Many times', 'Once' ) AS frequency
FROM products
JOIN order_items USING(product_id)
GROUP BY product_id, name
```

### 8 : Operator CASE

- operator CASE jest rozszerzeniem funkcji IF()

- składnia :

```
CASE
    WHEN warunek_1 THEN wartość_1
    WHEN warunek_2 THEN wartość_2
    // ...
    ELSE wartość-inne
```

END

- rozszerzając poprzedni przykład :

```
SELECT
    order_id,
    CASE
        WHEN YEAR( order_date) = YEAR( NOW()) THEN 'Active'
        WHEN YEAR( order_date) = YEAR( NOW()) -1 THEN 'Last Year'
        WHEN YEAR( order_date) < YEAR( NOW()) -1 THEN 'Archived'
        ELSE 'Future'
    END AS category
FROM orders
```

#### Zadanie 5 -----

```
SELECT
    CONCAT(first_name, ' ', last_name),
    points,
    CASE
        WHEN points > 3000 THEN 'Gold'
        WHEN points >= 2000 Then 'Silver'
        ELSE 'Bronze'
    END AS category
FROM customers
ORDER BY points DESC
```

---

## Secton 3 : Widoki

\*\*Tabela\*\*: Tabela jest głównym sposobem do przechowywania danych i informacji w RDBMS. Tabela jest zbiorem powiązanych wpisów danych (rekordów) i składa się z kolumn i wierszy.

\*\*Widok\*\*: widok to wirtualna tabela, której zawartość jest definiowana przez zapytanie. O ile widok nie jest indeksowany, widok nie istnieje jako przechowywany zestaw wartości danych w bazie danych. Zalety w stosunku do tabeli to:

1. Możemy łączyć kolumny / wiersze z wielu tabel lub innego widoku i mieć skonsolidowany widok.
2. Widoki mogą służyć jako mechanizmy bezpieczeństwa, umożliwiając użytkownikom dostęp do danych za pośrednictwem widoku, bez przyznawania użytkownikom uprawnień do bezpośredniego dostępu do bazowych tabel widoku.
3. Działa jako abstrakcyjna warstwa dla dalszych systemów, więc żadna zmiana w schemacie nie jest ujawniana, a zatem nie ma to wpływu na systemy niższego szczebla.

### Lesson 1 : Tworzenie widoków

#### Składnia :

```
CREATE VIEW `nazwa widoku` AS
SELECT
// ....
WHERE .....
```

#### Przykład

```
USE sql_invoicing;
```

```
CREATE VIEW sales_by_client AS
SELECT
    c.client_id,
    c.name,
    SUM(invoice_total) AS total_sales
FROM clients c
JOIN invoices i USING(client_id)
GROUP BY client_id, name
```

- widok jest widoczny w lewym panelu po odświeżeniu

- tak skonstrowany widok moze byc uzywany w innych zapytaniach, podobnie jak tablice

```
SELECT *
FROM sales_by_client
ORDER BY total_sales

- lub w złączeniach
// ...
JOIN clients USING( client_id)
```

#### Zadanie 1 -----  
W bazie sql\_invoicing stwórz widok, który dla każdego klienta (clients) będzie pokazywał jego Bilans :  
\*\*client\_balance\*\* :  
- client\_id,  
- name,  
- balance = invoice\_total - payment\_total

`Rozwiązanie :`

```
CREATE VIEW clients_balance AS
SELECT
    c.client_id,
    c.name,
    SUM( invoice_total - payment_total) AS balance
FROM clients c
JOIN invoices i USING(client_id)
GROUP BY client_id, name;
```

- mozemy uzyc widoku w zapytaniu

```
SELECT *
FROM clients_balance
WHERE balance > 500 -- warunek filtrujący > nie potrzeba klauzuli HAVING
ORDER BY balance DESC
```

## 2 : Modyfikacja i usuwanie widoków

- usuwanie widoku : DROP VIEW clients\_balance
  - następnie stworzy widok na nowo - w zmienionej formie
- lepiej CREATE OR REPLACE VIEW sales\_by\_client AS ...
- dobra praktyka : przechowuje widoki w systemie kontroli wersji, tak jak inny kod źródłowy

## 3 : Updateable views

- widoków mozna uzywac nie tylko do zapytań ale takze do poleceń DML : INSERT, UPDATE oraz DELETE.

- dotyczy to widoków które NIE zawierają :
  - DISTINCT
  - Funkcji agregujących (MIN, MAX, COUNT etc...)
  - GROUP BY / HAVING
  - UNION
- przykład : tablica invoices[]  
USE sql\_invoicing;

```
CREATE OR REPLACE VIEW invoices_with_balance AS
SELECT
    invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_total - payment_total AS balance,
```

```
        invoice_date,  
        due_date,  
        payment_date  
    FROM invoices  
    WHERE (invoice_total - payment_total) > 0  
  
- teraz mozemy uzywac widoku w SELECT, DELETE lub UPDATE  
DELETE FROM invoices_with_balance  
WHERE invoice_id = 1  
  
UPDATE invoices_with_balance  
SET due_date = DATE_ADD(due_date, INTERVAL 2 DAY)  
WHERE invoice_id = 2  
  
- uwaga na próby wstawiania poprzez Widok - widok musi zawierac w sobie wszystkie wymagane kolumny tabeli na jakiej jest oparty.  
  
### 4 : Klauzula WITH OPTION CHECK  
- w przypadku gdybyśmy w poprzednim przykładzie dla Klienta o client_id = 2 ustawili payment_total jako równe invoice_total  
    UPDATE invoices_with_balance  
    SET payment_total = invoice_total  
    WHERE invoice_id = 2  
- to klient o id = 2 zniknie nam z widoku, co jest domyślnym zachowaniem (widok ma filtr balance > 0)  
- klient o id = 2 nie zniknie natomiast z tablicy, na której oparty jest widok  
- takie domyślne zachowanie widoku można zmodyfikować stosując **WITH CHECK OPTION** na końcu kwerendy tworzącej widok :  
- Przykład :  
    CREATE OR REPLACE VIEW invoices_with_balance AS  
    // ...  
    WHERE (invoice_total - payment_total) > 0  
    WITH CHECK OPTION  
- UPDATE via widok nie zostanie wykonany - zostanie zgłoszony błąd  
-----  
## Section 4 : Złożone zapytania  
### Lesson 1 : Wprowadzenie  
- odtworzenie początkowej bazy danych  
  
### 2 : Subqueries  
- Zadanie : znajdź produkt droższy niż Lettuce (id = 3)  
    USE sql_store;  
  
    SELECT *  
    FROM products  
    WHERE unit_price > (  
        SELECT unit_price  
        FROM products  
        WHERE product_id = 3  
    )  
  
##### Zadanie 1 -----  
W bazie sql_hr, znajdź pracowników, którzy zarabiają więcej niż średnie wynagrodzenie w firmie.  
Rozwiązanie :  
    USE sql_hr;  
  
    SELECT *  
    FROM employees  
    WHERE salary > (  
        SELECT AVG(salary)  
        FROM employees  
    )
```

### ### 3 : Operator IN

- Operator IN () - przynależność do zbioru

#### #### Przykład

- Znajdź produkt, który nigdy nie był zamawiany.

```
USE sql_store;
```

```
SELECT *
FROM products
WHERE product_id NOT IN (
    SELECT DISTINCT product_id
    FROM order_items
)
```

#### #### Wydajność operatora IN (Opcjonalnie)

- IN jest zazwyczaj przechowywany w silniku DB w bazie podręcznej - właściwsze dla mniejszych zbiorów

-

#### #### Zadanie 2 -----

W bazie sql\_invoicing znajdź klientów, którzy nigdy nie mieli wystawionej faktury.

Rozwiązanie :

```
USE sql_invoicing;
```

```
SELECT *
FROM clients
WHERE client_id NOT IN (
    SELECT DISTINCT client_id
    FROM invoices
)
```

### ### 4 : Subqueries vs Joins

- można zastosować dwa kryteria :

- czytelność kodu - wybierz takie rozwiązanie, które lepiej pasuje do pytania "biznesowego"

- wydajność - dla dużych zbiorów konstrukcja IN/ NOT IN + subquery, **\*\*NIE JEST\*\* dobrym rozwiązaniem**

#### #### Zadanie 3 -----

W bazie sql\_store znajdź klientów, którzy kiedykolwiek zamówili produkt Lettuce (id = 3). Zapytanie skonstruuj z użyciem a) Subquery oraz b) JOIN.

Zwróć customer\_id, first\_name, last\_name

**\*\*Rozwiązanie a) Subquery\*\***

```
USE sql_store;
```

```
SELECT *
FROM customers
WHERE customer_id IN (
    SELECT o.customer_id
    FROM order_items oi
    JOIN orders o USING( order_id )
    where product_id = 3
)
```

**\*\*Rozwiązanie b) JOIN\*\***

```
SELECT DISTINCT
    customer_id,
    first_name,
    last_name
FROM customers c
JOIN orders o USING(customer_id)
JOIN order_items oi USING(order_id)
```

```
WHERE oi.product_id = 3

- dyskusja : które szybsze lub czytelniejsze ?
- raczej b)

### 5 : Operator EXISTS/ NOT EXISTS
- jest w niektórych przypadkach alternatywą dla IN + Subqueries
- operator IN produkuje zbiór znalezionych wartości, jeśli zbiór jest zbyt duży to :
  1. zbiór może nie zmieścić się w cache i zapytanie będzie wykonywane wielokrotnie
  2. w krytycznym przypadku mwykonywanie zapytania (subquery) może zostać przerwane
- podstawowym pytaniem jest : jak duży zbiór wartości zwraca Subquery ?

#### Przykład - znajdź klientów którzy mają (jakąś) fakturę
USE sql_invoicing;

SELECT *
FROM clients c
WHERE EXISTS (
    SELECT client_id -- szukamy jakikolwiek rekord dla którego będzie
    FROM invoices -- prawdziwy filtr WHERE w Subqu.
    WHERE client_id = c.client_id
)

alternatywnie :
SELECT *
FROM clients
WHERE client_id IN (
    SELECT DISTINCT client_id
    FROM invoices
)
-----#
#### Zadanie 4 -----
W bazie sql_store znajdź produkty, które nigdy nie były zamówione.
Zakładamy bardzo duży zbiór produktów.
Rozwiązanie :
USE sql_store;

SELECT *
FROM products p
WHERE NOT EXISTS (
    SELECT product_id
    FROM order_items
    WHERE product_id = p.product_id
)

### 6 : Subqueries w klauzuli SELECT
- Subqueries można używać nie tylko w klauzuli WHERE ale także w klauzulach SELECT i FROM
-
#### Przykład - znajdź różnice pomiędzy średnią wartością faktur a daną fakturą
USE sql_invoicing;

SELECT
    invoice_id,
    invoice_total,
    ( SELECT AVG(invoice_total)
        FROM invoices ) AS invoice_average,
    invoice_total - (SELECT invoice_average) AS different -- zapis skrócony pełnego
wywołania
    FROM invoices

- w powyższym nie możemy się bezpośrednio odwołać do invoice_average bo jest to kolumna
wyliczana z wyrażenia - musimy ponownie wykonac SELECT
- konstrukcja ( SELECT invoice_average ) jest zapisem skróconym pełnego wywołania
-
```

#### Zadanie 5 -----  
Rozwiązanie :  

```
SELECT
    client_id,
    name,
    (
        SELECT SUM(invoice_total)
        FROM invoices
        WHERE client_id = c.client_id
    ) AS total_sales,
    (
        SELECT AVG(invoice_total)
        FROM invoices
    ) AS average_invoices,
    (SELECT total_sales - average_invoices) AS difference
FROM clients c
```

### 7 : Subqueries w klauzuli FROM

- zapytanie z poprzedniej lekcji produkuje nam wynik - tak naprawdę tablicę tymczasową.  
Tą balicę możemy uzyc w klauzuli FROM i z niej pobierac dane, filtrowac, sortowac etc.

#### Przykład

```
SELECT *
FROM (
    SELECT
        client_id,
        name,
        (
            SELECT SUM(invoice_total)
            FROM invoices
            WHERE client_id = c.client_id
        ) AS total_sales,
        (
            SELECT AVG(invoice_total) FROM invoices
        ) AS average,
        (
            SELECT total_sales - average
        ) AS difference
    FROM clients c
) AS sales_summary -- alias jest obowiązkowy
WHERE total_sales IS NOT NULL
- przy tak skomplikowanym zapytaniu o wiele lepszym rozwiązaniem byłoby uzycie widoku,
zamiast subquery
- subquery zostawmy dla prostych incydentalnych zapytań
```

---

## Section 5 : Tworzenie i zarządzanie bazą danych

### Lesson 0 : Zarządzanie bazami - powtórka z wcześniejszej prezentacji  
#### Utworzenie nowej bazy danych

Wszystkie dane chcemy trzymać w tabelach w pewnej bazie danych. W celu jej stworzenia możemy wykorzystać polecenie:

```
CREATE DATABASE 'nazwa_bazy';
gdzie nazwa bazy danych jest dowolną nazwą, którą ma przyjąć baza danych.
```

- jeśli komenda jest wykonywana w skrypcie to lepiej sprawdzic czy baza juz nie istnieje

```
CREATE DATABASE IF NOT EXISTS sql_store2;
USE sql_store2;
```

#### Usunięcie bazy danych

Usuwanie bazy danych jest możliwe za pomocą instrukcji:

```
DROP DATABASE 'nazwa_bazy_do_usunięcia';
```

- jeśli chcemy byc pewni, ze usuwamy istniejącą bazę danych, uzywamy konstrukcji :  
DROP DATABASE IF EXISTS 'nazwa\_bazy';

#### Praca z bazą danych

W związku z tym, że na serwerze bazodanowym może istnieć wiele baz danych, jest możliwość wyboru bazy danych, na której będziemy pracować. Służy do tego polecenie use, np.:

```
USE 'nazwa_bazy_danych';  
Należy pamiętać, że wejście do bazy danych jest możliwe tylko i wyłącznie, jeśli użytkownik ma odpowiednie prawa dostępowe.
```

#### Tworzenie tablicy w bazie danych

```
USE sql_store2;
```

```
CREATE TABLE IF NOT EXISTS customers (  
    customer_id INT PRIMARY KEY AUTO_INCREMENT,  
    first_name VARCHAR(50) NOT NULL,  
    points      INT NOT NULL DEFAULT 0,  
    email       VARCHAR(255) NOT NULL UNIQUE  
)
```

#### Zmiana zawartości tablicy / usunięcie tablicy

```
ALTER TABLE customers  
    ADD last_name VARCHAR(50) NOT NULL AFTER first_name,  
    MODIFY first_name VARCHAR(55) DEFAULT '',  
    DROP points; -- lepiej explicite zaznaczy DROP COLUMN
```

#### Tworzenie relacji między tablicami

- najpierw stworzymy drugą tablicę
- DROP TABLE IF EXISTS orders;

```
CREATE TABLE orders (  
    order_id    INT PRIMARY KEY,  
    customer_id INT NOT NULL,  
    FOREIGN KEY fk_orders_customers(customer_id)  
        REFERENCES customers (customer_id)  
        ON UPDATE CASCADE  
        ON DELETE NO ACTION  
)
```

## Lesson 1 : Tworzenie fizycznego modelu danych

- kolejność modelowania bazy danych
- 1. Zrozumienie wymagań biznesowych - Analiza biznesowa
- 2. Model koncepcyjny
- 3. Model logiczny
- 4. Model fizyczny

- 
- 5. Wygenerowanie skryptu DDL + implementacja na serwerze

## 2 Normalizacja bazy danych - trzy postacie normalne 1NF -2NF -3NF

#### Pierwsza postać normalna - przykład

- aktualny model fizyczny nie spełnia postulatu 1NF - atrybut tags w tablicy courses[] może zawierać wiele wartości

- dodanie tabeli tagów do tablicy courses
  - tag\_id TINYINT
  - name VARCHAR(50)
- rozwiążanie relacji N : N - dodanie tablicy course\_tags[]
  - course\_id (fk1) INT
  - tag\_id TINYINT
- trzeba usunąć atrybut tags z courses[]

#### Druga postać normalna - przykład

- atrybut instructor w tablicy courses[] pojęciowo nie jest związany z Kursami

- dodanie nowej tablicy instructors[]

- instructor\_id SMALLINT, PK, AI, NN

- name VARCHAR(50)

- dodaj relację 1 : N (child > parent)

- usuń atrybut instructor z tablicy courses[]

#### Trzecia posta normalna - przykład

- payment\_total - invoicing\_total = balanc

- kolumna balance jest nadmiarowa > moze byc wyliczona z dwóch pozostałych

#### Zadanie -----

System rezerwacji lotów

### 6 : Zarządzanie tablicami etc w bazie danych z poziomu SQL

Do samodzielnego przestudiowania :

[https://java.pl.sdacademy.pro/e-podrecznik/bazy\\_danych\\_sql/sql/ddl/](https://java.pl.sdacademy.pro/e-podrecznik/bazy_danych_sql/sql/ddl/)