

Lekcja

Temat: Funkcje związane z czasem, datą, operatorami łańcuchowymi

Funkcje daty i czasu

Link do dokumentacji MySQL:

<https://dev.mysql.com/doc/refman/8.4/en/date-and-time-functions.html>

Metoda	Wyjaśnienie	Przykład SQL	Wynik
ADDDATE()	Dodaje interwał do daty	SELECT ADDDATE('2024-01-01', INTERVAL 5 DAY);	2024-01-06
ADDTIME()	Dodaje czas	SELECT ADDTIME('10:00:00','02:30:00');	12:30:00
CONVERT_TZ()	Konwersja strefy czasowej	SELECT CONVERT_TZ('2024-01-01 12:00','UTC','Europe/Warsaw');	2024-01-01 13:00
CURDATE()	Bieżąca data	SELECT CURDATE();	2025-11-10
CURTIME()	Bieżący czas	SELECT CURTIME();	np. 14:22:01
DATE()	Zwraca część datową	SELECT DATE('2024-01-01 10:00:00');	2024-01-01
DATE_ADD()	Dodaje interwał do daty	SELECT DATE_ADD('2024-01-01', INTERVAL 1 MONTH);	2024-02-01
DATE_FORMAT()	Formatuje datę	SELECT DATE_FORMAT('2024-01-15','%d-%m-%Y');	15-01-2024
DATE_SUB()	Odejmuje interwał	SELECT DATE_SUB('2024-01-10', INTERVAL 3 DAY);	2024-01-07
DATEDIFF()	Różnica między datami	SELECT DATEDIFF('2024-02-01','2024-01-01');	31

DAY()	Dzień miesiąca	SELECT DAY('2024-01-15');	15
DAYNAME()	Nazwa dnia	SELECT DAYNAME('2024-01-15');	Tuesday
DAYOFMONTH()	Dzień miesiąca	SELECT DAYOFMONTH('2024-01-15');	15
DAYOFWEEK()	Numer dnia tyg. (1=nd)	SELECT DAYOFWEEK('2024-01-15');	3
DAYOFYEAR()	Dzień roku	SELECT DAYOFYEAR('2024-01-15');	15
EXTRACT()	Wyodrębnia część daty	SELECT EXTRACT(YEAR FROM '2024-01-15');	2024
FROM_DAYS()	Dni → data	SELECT FROM_DAYS(750000);	2044-01-22
FROM_UNIXTIME()	UNIX → data	SELECT FROM_UNIXTIME(1700000000);	2023-11-14 22:13:20
HOUR()	Pobiera godzinę	SELECT HOUR('12:45:00');	12
LAST_DAY()	Ostatni dzień miesiąca	SELECT LAST_DAY('2024-02-10');	2024-02-29
MAKEDATE()	Tworzy datę z dnia roku	SELECT MAKEDATE(2024,32);	2024-02-01
MAKETIME()	Tworzy czas	SELECT MAKETIME(10,20,30);	10:20:30
MICROSECOND()	Mikrosekundy	SELECT MICROSECOND('10:00:00.123456');	123456

MINUTE()	Minuta	SELECT MINUTE('12:45:30');	45
MONTH()	Numer miesiąca	SELECT MONTH('2024-05-10');	5
MONTHNAME()	Nazwa miesiąca	SELECT MONTHNAME('2024-05-10');	May
NOW()	Aktualny datetime	SELECT NOW();	2025-11-10 14:20:xx
PERIOD_ADD()	Dodaje miesiące do YYYYMM	SELECT PERIOD_ADD(202401,2);	202403
PERIOD_DIFF()	Ilość miesięcy między okresami	SELECT PERIOD_DIFF(202402,202401);	1
QUARTER()	Kwartał	SELECT QUARTER('2024-05-10');	2
SEC_TO_TIME()	Sekundy → czas	SELECT SEC_TO_TIME(3661);	01:01:01
SECOND()	Sekundy	SELECT SECOND('12:45:59');	59
STR_TO_DATE()	Tekst → data	SELECT STR_TO_DATE('31-01-2024','%d-%m-%Y');	2024-01-31
SUBTIME()	Odejmuje czas	SELECT SUBTIME('10:00:00','01:30:00');	08:30:00
SYSDATE()	Czas wykonania	SELECT SYSDATE();	2025-11-10...

TIME()	Czas z datetime	SELECT TIME('2024-01-01 12:30:45');	12:30:45
TIME_FORMAT()	Formatuje czas	SELECT TIME_FORMAT('12:30:45','%H:%i');	12:30
TIME_TO_SEC()	Czas → sekundy	SELECT TIME_TO_SEC('01:00:00');	3600
TIMEDIFF()	Różnica czasu	SELECT TIMEDIFF('12:00:00','10:00:00');	02:00:00
TIMESTAMP()	Tworzy datetime	SELECT TIMESTAMP('2024-01-01');	2024-01-01 00:00:00
TIMESTAMPADD()	Dodaje interwał	SELECT TIMESTAMPADD(HOUR,2,'2024-01-01 10:00');	2024-01-01 12:00
TIMESTAMPDIFF()	Różnica datetime	SELECT TIMESTAMPDIFF(DAY,'2024-01-01','2024-01-10');	9
TO_DAYS()	Data → dni od roku 0	SELECT TO_DAYS('2024-01-01');	739252
TO_SECONDS()	Data → sekundy od roku 0	SELECT TO_SECONDS('2024-01-01');	64092288000
UNIX_TIMESTAMP()	Aktualny UNIX time	SELECT UNIX_TIMESTAMP();	np. 1768060000
UTC_DATE()	Data UTC	SELECT UTC_DATE();	2025-11-10

UTC_TIME()	Czas UTC	SELECT UTC_TIME();	13:14:xx
UTC_TIMESTAMP()	Datetime UTC	SELECT UTC_TIMESTAMP();	2025-11-10 13:14:xx
WEEK()	Numer tygodnia	SELECT WEEK('2024-01-10');	1
WEEKDAY()	Dzień tyg. (0=pon)	SELECT WEEKDAY('2024-01-10');	3
WEEKOFYEAR()	Tydzień ISO	SELECT WEEKOFYEAR('2024-01-10');	2
YEAR()	Rok	SELECT YEAR('2024-01-10');	2024
YEARWEEK()	Rok + tydzień	SELECT YEARWEEK('2024-01-10');	202402

UTC (Uniwersalny Czas Koordynowany) to światowy standard czasu atomowego, który służy jako podstawa do ustalania lokalnego czasu w różnych strefach czasowych. Polska znajduje się w strefie czasowej UTC+1 (czas środkowoeuropejski, CET) zimą i UTC+2 (czas środkowoeuropejski letni, CEST) latem, a lokalny czas w Polsce jest o 1 lub 2 godziny późniejszy od czasu UTC.

- Co to jest UTC:
 - UTC to międzynarodowy standard czasu, który jest niezależny od ruchu obrotowego Ziemi i oparty na bardzo precyzyjnym czasie atomowym.

- Jest to punkt odniesienia, taki sam na całym świecie, do którego dodaje się lub od którego odejmuje się czas, aby uzyskać lokalny czas dla danej strefy czasowej.

- **UTC w Polsce:**

- Polska leży w strefie czasowej UTC+1 (czas zimowy) lub UTC+2 (czas letni).
- Czas zimowy (CET): Obowiązuje od ostatniej niedzieli października do ostatniej niedzieli marca. Czas lokalny w Polsce jest o 1 godzinę późniejszy niż UTC. (np. jeśli UTC to 12:00, w Polsce jest 13:00).
- Czas letni (CEST): Obowiązuje od ostatniej niedzieli marca do ostatniej niedzieli października. Czas lokalny w Polsce jest o 2 godziny późniejszy niż UTC. (np. jeśli UTC to 12:00, w Polsce jest 14:00).

Zastosowania:

- Programowanie - przechowywanie dat i czasu w bazach danych
- Lotnictwo - koordynacja lotów międzynarodowych
- Internet - synchronizacja serwerów
- Telekomunikacja - koordynacja transmisji
- Nauka - precyzyjne pomiary czasu

W praktyce: Gdy widzisz znacznik czasu typu `2025-11-11T14:30:00Z`, litera "Z" na końcu oznacza właśnie UTC (od "Zulu time" - wojskowego określenia UTC).

Przykłady:

- Polska: UTC+1 (zimą) lub UTC+2 (latem)
- Nowy Jork: UTC-5 (zimą) lub UTC-4 (latem)
- Tokio: UTC+9
- Londyn: UTC+0 (zimą) lub UTC+1 (latem)

Funkcje i operatory łańcuchowe

Link do dokumentacji MySQL:

<https://dev.mysql.com/doc/refman/8.4/en/string-functions.html>

Metoda	Wyjaśnienie	Przykład	Wynik
ASCII()	Zwraca kod ASCII pierwszego znaku	SELECT ASCII('A');	65
BIN()	Zwraca liczbę w postaci binarnej	SELECT BIN(10);	1010
BIT_LENGTH()	Zwraca długość napisu w bitach	SELECT BIT_LENGTH('ABC');	24
CHAR()	Zwraca znak odpowiadający podanemu kodowi ASCII	SELECT CHAR(65);	'A'
CHAR_LENGTH()	Liczba znaków (nie bajtów)	SELECT CHAR_LENGTH('Łódź');	4
CHARACTER_LENGTH()	To samo co CHAR_LENGTH()	SELECT CHARACTER_LENGTH('Test');	4
CONCAT()	Łączy napisy	SELECT CONCAT('A', 'B', 'C');	'ABC'
CONCAT_WS()	Łączy napisy z separatorem	SELECT CONCAT_WS('-', 'A','B','C');	'A-B-C'

ELT()	Zwraca element listy na indeksie (1-based)	SELECT ELT(2,'jeden','dwa','trzy');	'dwa'
EXPORT_SET()	Zamienia liczby bitowe na tekst ON/OFF	SELECT EXPORT_SET(5, 'ON', 'OFF', ',', 4);	ON,OFF,ON,OFF
FIELD()	Zwraca pozycję pierwszego argumentu w liście	SELECT FIELD('kot','pies','kot','mysz');	2
FIND_IN_SET()	Pozycja elementu w liście CSV	SELECT FIND_IN_SET('B', 'A,B,C');	2
FORMAT()	Formatuje liczbę z przecinkami	SELECT FORMAT(12345.678, 2);	'12,345.68'
FROM_BASE64()	Dekoduje Base64	SELECT FROM_BASE64('SGVsbG8=');	'Hello'
HEX()	Zamienia liczbę lub tekst na hex	SELECT HEX('ABC');	414243
INSERT()	Wstawia podciąg w podaną pozycję, zastępując określoną liczbę znaków	SELECT INSERT('abcdef', 3, 2, 'XYZ');	'abXYZef'
INSTR()	Pozycja pierwszego wystąpienia podciągu	SELECT INSTR('abcabc','ca');	3
LCASE()	To samo co LOWER() – zamienia na małe litery	SELECT LCASE('Test');	'test'

LEFT()	Zwraca określoną liczbę znaków od lewej	SELECT LEFT('abcdef', 3);	'abc'
LENGTH()	Długość napisu w bajtach	SELECT LENGTH('ABC');	3
LIKE	Sprawdza dopasowanie wzorca	SELECT 'Ala' LIKE 'A%';	1
LOAD_FILE()	Wczytuje zawartość pliku (jeśli SQL ma dostęp)	SELECT LOAD_FILE('/path/file.txt');	<i>treść pliku</i>
LOCATE()	Pozycja podciągu (jak INSTR, ale kolejność argumentów odwrotna)	SELECT LOCATE('b','abc');	2
LOWER()	Zamienia na małe litery	SELECT LOWER('TEST');	'test'
LPAD()	Uzupełnia z lewej do zadanej długości	SELECT LPAD('7', 3, '0');	'007'
LTRIM()	Usuwa spacje z lewej	SELECT LTRIM(' test');	'test'
MAKE_SET()	Zwraca listę elementów pasujących do bitów liczby	SELECT MAKE_SET(5,'A','B','C');	'A,C'
MATCH() AGAINST()	Pełnotekstowe wyszukiwanie	SELECT MATCH(text) AGAINST('kot');	<i>ocena dopasowania</i>

MID()	Alias SUBSTRING()	SELECT MID('abcdef', 2, 3);	'bcd'
NOT LIKE	Odwrotność LIKE	SELECT 'Ala' NOT LIKE 'K%';	1
NOT REGEXP	Odwrotność REGEXP	SELECT 'abc' NOT REGEXP '^ [0-9]+\$';	1
OCT()	Zamienia liczbę na system ósemkowy	SELECT OCT(15);	'17'
OCTET_LENGTH()	Alias LENGTH()	SELECT OCTET_LENGTH('ABC');	3
ORD()	Kod ASCII pierwszego znaku	SELECT ORD('A');	65
POSITION()	Alias LOCATE()	SELECT POSITION('a' IN 'banan');	2
QUOTE()	Zwraca tekst w bezpiecznej formie (escape)	SELECT QUOTE("Ala's cat");	'Ala\'s cat'
REGEXP	Dopasowanie wyrażenia regularnego	SELECT 'abc123' REGEXP '[0-9]+';	1
REGEXP_INSTR()	Pozycja dopasowania regexu	SELECT REGEXP_INSTR('abc123','[0-9]+');	4
REGEXP_LIKE()	Czy pasuje regex	SELECT REGEXP_LIKE('test123','[a-z]+');	1

REGEXP_REPLACE()	Zamienia dopasowane fragmenty	SELECT REGEXP_REPLACE('a1b2c3','[0-9]','X');	'aXbXcX'
REGEXP_SUBSTR()	Zwraca fragment pasujący do regexu	SELECT REGEXP_SUBSTR('abc123','[0-9]+');	'123'
REPEAT()	Powtarza tekst	SELECT REPEAT('A',3);	'AAA'
REPLACE()	Podmienia tekst	SELECT REPLACE('ala ma kota','a','X');	'XIX mX kotX'
REVERSE()	Odwraca napis	SELECT REVERSE('kota');	'atok'
RIGHT()	Znaki od prawej	SELECT RIGHT('abcdef', 2);	'ef'
RLIKE	Alias REGEXP	SELECT 'abc' RLIKE '[a-z]+';	1
RPAD()	Uzupełnia napis z prawej	SELECT RPAD('A', 4, '.');	'A...'
RTRIM()	Usuwa spacje z prawej	SELECT RTRIM('test ');	'test'
SOUNDEX()	Kod fonetyczny słów	SELECT SOUNDEX('Robert');	'R163'
SOUNDS LIKE	Porównanie brzmienia	SELECT 'Robert' SOUNDS LIKE 'Rupert';	1
SPACE()	Generuje spacje	SELECT SPACE(5);	' '
STRCMP()	Porównuje napisy	SELECT STRCMP('abc','abd');	-1

SUBSTR()	Podciąg (alias SUBSTRING)	SELECT SUBSTR('abcdef',2,3);	'bcd'
SUBSTRING()	Podciąg	SELECT SUBSTRING('abcdef',3);	'cdef'
SUBSTRING_INDEX()	Podciąg do N-tego separatora	SELECT SUBSTRING_INDEX('a,b,c',';',2);	'a,b'
TO_BASE64()	Kodowanie Base64	SELECT TO_BASE64('Hello');	'SGVsbG8='
TRIM()	Usuwa spacje z obu stron	SELECT TRIM(' test ');	'test'
UCASE()	Alias UPPER()	SELECT UCASE('abc');	'ABC'
UNHEX()	Hex → tekst	SELECT UNHEX('414243');	'ABC'
UPPER()	Zamienia na wielkie litery	SELECT UPPER('kot');	'KOT'
WEIGHT_STRING()	Zwraca wewnętrzną wagę znaków (techniczne)	SELECT WEIGHT_STRING('A');	(hex bajty)

Lekcja

Temat: Właściwości kolumn (pola) w MySQL: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, DEFAULT, CHECK, AUTO_INCREMENT, ENUM, COMMENT. ERD (Diagram związków encji ang. Entity Relationship Diagram)

W MySQL możesz nałożyć **wiele rodzajów właściwości (constraints)** na pojedynczą kolumnę albo na kilka kolumn naraz, żeby wymusić reguły zachowania danych.

✓ 1. NOT NULL

Kolumna **nie może przyjmować wartości NULL**.
Wymusza, że musisz zawsze podać wartość.

Przykład:

```
CREATE TABLE osoby (  
  id INT NOT NULL,  
  imie VARCHAR(100) NOT NULL  
);
```

Wyjaśnienie:

- **imie** i **id** **musi** być podane.

✓ 2. UNIQUE

Wymusza **unikalne wartości** w kolumnie — nie mogą się powtarzać.

Przykład:

```
CREATE TABLE klienci (  
  id INT PRIMARY KEY,  
  email VARCHAR(255) UNIQUE  
);
```

Wyjaśnienie:

Dwa takie same maile → ❌ błąd.

Można też ustawić UNIQUE na **kilka kolumn naraz**:

UNIQUE (uczen_id, kurs_id)

✅ 3. PRIMARY KEY

- jednoznacznie identyfikuje każdy wiersz (unikalny),
- automatycznie ma **UNIQUE + NOT NULL**.

Przykład:

```
CREATE TABLE produkty (  
    produkt_id INT PRIMARY KEY,  
    nazwa VARCHAR(100)  
);
```

Możesz też zrobić klucz **złożony z kilku kolumn**:

PRIMARY KEY (zamowienie_id, produkt_id)

✅ 4. FOREIGN KEY

Łączy tabele — kolumna musi wskazywać na wartość z innej tabeli.

Przykład:

```
CREATE TABLE zamowienia (  
    id INT PRIMARY KEY  
);
```

```
CREATE TABLE produkty_w_zamowieniu (  
    zamowienie_id INT,  
    produkt_id INT,  
    FOREIGN KEY (zamowienie_id) REFERENCES zamowienia(id)  
);
```

Nie można dodać produktu do zamówienia, które nie istnieje.

✅ 5. DEFAULT

Ustawia **wartość domyślną**, jeśli użytkownik nie poda swojej.

Przykład:

```
CREATE TABLE artykuły (  
    id INT PRIMARY KEY,  
    status VARCHAR(20) DEFAULT 'aktywny'  
);
```

Jeśli nie podasz statusu → automatycznie będzie „aktywny”.

✅ 6. CHECK

Wymusza spełnienie **logicznego warunku**.

Przykład:

```
CREATE TABLE pracownicy (  
  id INT PRIMARY KEY,  
  wiek INT CHECK (wiek >= 18 AND wiek <= 65)  
);
```

Próba dodania `wiek = 10` → ❌ błąd.

✅ 7. AUTO_INCREMENT

Automatycznie zwiększa wartość w kolumnie liczbowej przy każdym INSERT.

Przykład:

```
CREATE TABLE logi (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  opis VARCHAR(255)  
);
```

Dodajesz 5 logów → id będą: 1, 2, 3, 4, 5.

✅ 8. ENUM

Ogranicza wartości w kolumnie do **zamkniętej listy dopuszczalnych opcji**.

Przykład:

```
CREATE TABLE uzytkownicy (  
  id INT PRIMARY KEY,  
  plec ENUM('M', 'K', 'INNE') DEFAULT 'INNE'  
);
```

Próba zapisania `plec = 'ABC'` → ❌ błąd.

✅ 9. COMMENT

Pozwala dopisać **komentarz** do kolumny — bardzo przydatne przy dokumentowaniu schematu.

Przykład:

```
CREATE TABLE produkty (  
  id INT PRIMARY KEY,  
  cena DECIMAL(10,2) COMMENT 'Cena brutto w zł'  
);
```

W narzędziach typu phpMyAdmin, DBeaver zobaczysz komentarz przy kolumnie.

ERD — diagram związków encji

To graficzny sposób przedstawienia struktury bazy danych:

- jakie **tabele (encje)** istnieją,
- jakie mają **atrybuty (kolumny)**,
- jakie występują **relacje** między tabelami:

- 1:1
- 1:N
- N:M

ERD jest tworzony zanim powstanie baza danych, aby zaplanować jej strukturę.

Encja (Entity) = obiekt, który ma znaczenie w systemie i który chcesz zapisać w bazie.

Inaczej mówiąc:

👉 **Encja** = tabela w bazie danych

👉 **Atrybut** = kolumna w tabeli

Przykłady encji:

- **User** (użytkownik)
- **Product** (produkt)
- **Order** (zamówienie)
- **Invoice** (faktura)
- **Department** (dział firmy)

Każda encja ma klucz główny (Primary Key, PK) – unikalny identyfikator, np. id.

Tworzenie krok po kroku diagramu związków encji

Krok 1: Zidentyfikuj encje (tabele)

Krok 2: Określ atrybuty

Dla każdej encji określasz pola.

Przykład:

Customer

- id
- first_name
- last_name
- email

Krok 3: Ustal klucze główne

Każda encja ma PK:

Krok 4: Określ relacje między encjami

1) Relacja 1:1 (One to One)

Jeden rekord odpowiada dokładnie jednemu rekordowi w drugiej tabeli.

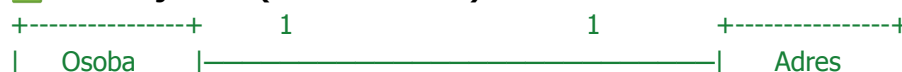
2) Relacja 1:N (One to Many)

Jeden klient może mieć wiele zamówień.

3) Relacja N:M (Many to Many)

Tworzy się tabelę pośredniczącą.

✓ 1. Relacja 1 : 1 (Osoba — Adres)

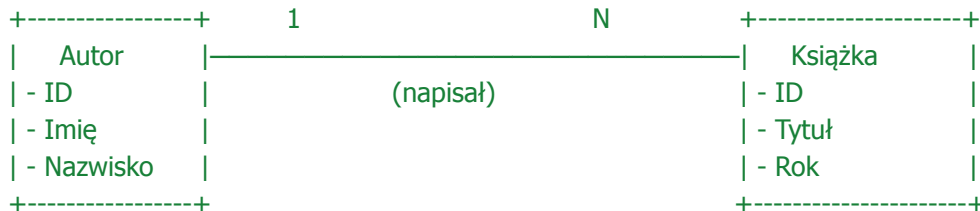


- ID	
- Imię	
- Nazwisko	
+-----+	

(ma)

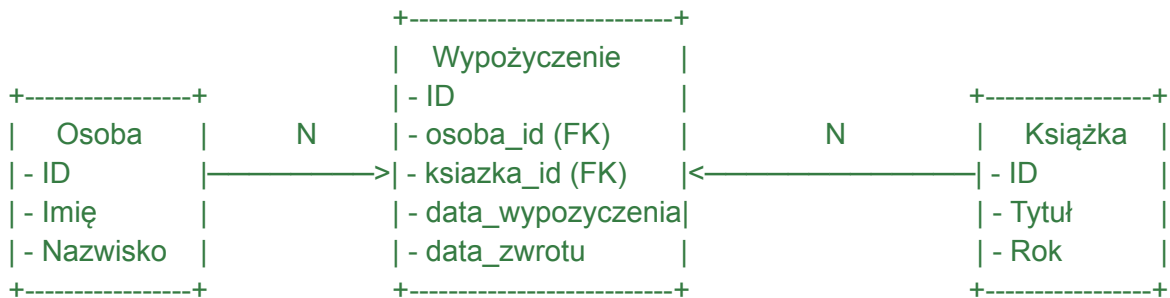
- ID	
- Ulica	
- Miasto	
+-----+	

✓ 2. Relacja 1 : N (Autor — Książka)



✓ 3. Relacja N : N (Osoba — Książka) przez tabelę Wypożyczenie

W MySQL/SQL relacja N:N **zawsze wymaga tabeli pośredniej**.



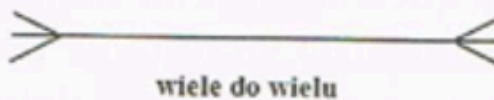
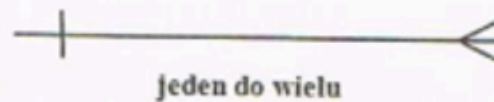
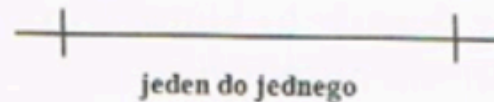
N : N

(wiele osób wypożycza wiele książek)

Opis reprezentacji graficznej stopnia związku został pokazany na rysunku

Rysunek

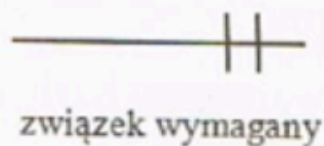
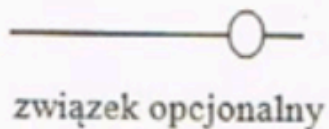
Graficzna reprezentacja
związków zachodzą-
cych między encjami



Opis reprezentacji graficznej opcjonalności związku został pokazany na rysunku

Rysunek

Graficzna reprezentacja
opcjonalności związku



Diagramy ERD możemy tworzyć za pomocą różnych notacji. Najpopularniejsze są diagramy w zapisie według Martina i Chena.

Lekcja

Temat: Group by

GROUP BY w **MySQL** służy do **grupowania rekordów**, które mają te same wartości w określonych kolumnach. Zazwyczaj używa się go **razem z funkcjami agregującymi**, takimi jak:

- **COUNT()** – zlicza ilość rekordów,
- **SUM()** – sumuje wartości,
- **AVG()** – liczy średnią,
- **MAX()** – zwraca wartość maksymalną,
- **MIN()** – zwraca wartość minimalną.

♦ Przykład praktyczny

```
CREATE TABLE orders (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  customer_name VARCHAR(50),  
  product_name VARCHAR(50),  
  quantity INT,  
  price DECIMAL(10, 2)  
);
```

```
INSERT INTO orders (customer_name, product_name, quantity, price)
VALUES
('Jan Kowalski', 'Laptop', 1, 3500.00),
('Anna Nowak', 'Mysz', 2, 50.00),
('Jan Kowalski', 'Mysz', 1, 50.00),
('Piotr Wiśniewski', 'Klawiatura', 1, 120.00),
('Anna Nowak', 'Laptop', 1, 3400.00),
('Jan Kowalski', 'Laptop', 1, 3500.00);
```

♦ **Przykład 1 – Suma wartości zamówień dla każdego klienta**

```
SELECT customer_name, SUM(price * quantity) AS total_spent
FROM orders
GROUP BY customer_name;
```

♦ **Przykład 2 – Ile produktów kupił każdy klient**

```
SELECT customer_name, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_name;
```

♦ **Przykład 3 – Średnia cena produktów kupionych przez każdego klienta**

```
SELECT customer_name, AVG(price) AS avg_price
FROM orders
GROUP BY customer_name;
```

♦ **Przykład 4 – Grupowanie po dwóch kolumnach (klient + produkt)**

```
SELECT customer_name, product_name, SUM(quantity) AS total_quantity  
FROM orders  
GROUP BY customer_name, product_name;
```

♦ **Przykład 5** GROUP BY z HAVING

Założmy, że chcemy zobaczyć **tylko tych klientów**, którzy **wydali łącznie więcej niż 1000 zł**.

```
SELECT customer_name, SUM(price * quantity) AS total_spent  
FROM orders  
GROUP BY customer_name  
HAVING SUM(price * quantity) > 1000;
```

♦ **Przykład 6** filtracja po liczbie zamówień

Chcemy zobaczyć klientów, którzy złożyli więcej niż jedno zamówienie:

```
SELECT customer_name, COUNT(*) AS total_orders  
FROM orders  
GROUP BY customer_name  
HAVING COUNT(*) > 1;
```

♦ Różnica między **WHERE** a **HAVING**

- **WHERE** filtruje **pojedyncze rekordy przed grupowaniem**,
- **HAVING** filtruje **całe grupy po agregacji**.

📌 Przykład błędu:

-- ❌ Nie zadziała:

```
SELECT customer_name, SUM(price)
FROM orders
WHERE SUM(price) > 1000
GROUP BY customer_name;
```

📌 Poprawnie:

```
SELECT customer_name, SUM(price)
FROM orders
GROUP BY customer_name
HAVING SUM(price) > 1000;
```

♦ Podsumowanie

- **GROUP BY** **grupuje** dane na podstawie wartości w kolumnach.
- Zazwyczaj używa się go z **funkcjami agregującymi** (**SUM**, **COUNT**, **AVG**, itp.).
- Może grupować po **jednej lub wielu kolumnach**.
- Często łączy się z **HAVING**, aby filtrować wyniki po agregacji (np. „pokaż tylko klientów, którzy wydali więcej niż 1000 zł”).

Lekcja

Temat: Kopie zapasowe i przywracanie w MySQL

Kopia zapasowa to zapis danych z bazy MySQL do pliku, aby można je było **odzyskać w razie:**

- awarii serwera,
- usunięcia danych,
- ataku (np. ransomware),
- błędu użytkownika.

Backupy w MySQL – metody i strategie

Wykonanie kopii zapasowej wykonuje się poleceniem mysqldump

1 mysqldump – backup logiczny

- odczytuje **strukturę bazy** (bazy, tabele, indeksy),
- odczytuje **dane z tabel**,
- zapisuje wszystko do **pliku tekstowego .sql**,
- w pliku są polecenia SQL (CREATE, INSERT).

♦ Kopia jednej bazy danych

mysqldump -u root -p nazwa_bazy > backup.sql

Co to znaczy:

- **-u root** → użytkownik MySQL
- **-p** → zapyta o hasło
- **nazwa_bazy** → baza do skopiowania
- **backup.sql** → plik kopii zapasowej

📌 Po wykonaniu masz plik, który zawiera **całą bazę**.

♦ **Kopia wszystkich baz**

mysqldump -u root -p --all-databases > all_backup.sql

Przywracanie bazy

♦ **Przywracanie do istniejącej bazy**

mysql -u root -p nazwa_bazy < backup.sql

♦ **Przywracanie nowej bazy**

mysql -u root -p < backup.sql

(MySQL sam utworzy bazę, jeśli polecenia CREATE DATABASE są w pliku)

Backup a cyberbezpieczeństwo

Kopie zapasowe:

- chronią przed **utratą danych**,
- pozwalają odzyskać dane po **ataku ransomware**,
- są elementem **polityki bezpieczeństwa**.

📌 Bez backupu = utrata danych na stałe.

♦ **mysqldump** - Zalety

- ✓ prosty
- ✓ przenośny
- ✓ idealny do nauki i małych baz

♦ **mysqldump** - Wady

- ✗ wolny przy dużych bazach
- ✗ brak prawdziwych backupów inkrementalnych

2 **Percona XtraBackup** – backup fizyczny

Percona XtraBackup to zaawansowane narzędzie do fizycznych kopii zapasowych MySQL/MariaDB.

- kopiuje pliki danych (InnoDB)
- działa bez zatrzymywania serwera
- używane w firmach i produkcji

Percona XtraBackup:

- kopiuje pliki **.ibd**, **.frm**, logi transakcji

- zapisuje je do katalogu backupu
- backup jest spójny (consistent)

Percona XtraBackup wymaga pełnej instalacji MySQL (nie XAMPP)

Lekcja

Temat: Widoki (VIEW), tabele tymczasowe (temporary tables) w MySQL. Wstęp do INDEX.

-- Tabela klientów

```
CREATE TABLE klienci (  
  id_klienta INT PRIMARY KEY AUTO_INCREMENT,  
  imie VARCHAR(50) NOT NULL,  
  nazwisko VARCHAR(50) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL  
);
```

-- Tabela zamówień

```
CREATE TABLE zamowienia (  
  id_zamowienia INT PRIMARY KEY AUTO_INCREMENT,  
  id_klienta INT NOT NULL,  
  data DATE NOT NULL,  
  kwota DECIMAL(10, 2) NOT NULL,  
  FOREIGN KEY (id_klienta) REFERENCES klienci(id_klienta)
```

```
);
```

-- Wstawianie klientów

```
INSERT INTO klienci (imie, nazwisko, email) VALUES  
( 'Jan', 'Kowalski', 'jan.kowalski@example.com'),  
( 'Anna', 'Nowak', 'anna.nowak@example.com'),  
( 'Piotr', 'Wiśniewski', 'piotr.wisniewski@example.com');
```

-- Wstawianie zamówień (mieszanka dat: nowe i stare)

```
INSERT INTO zamowienia (id_klienta, data, kwota) VALUES  
(1, '2024-12-01', 150.00),  
(1, '2025-06-15', 200.00),  
(2, '2025-11-20', 300.00),  
(2, '2024-05-10', 100.00),  
(3, '2026-01-05', 250.00);
```

Widoki (**VIEW**) w MySQL to **wirtualne tabele**, które **nie przechowują danych**, tylko **zapamiętują zapytanie SQL**.

Za każdym razem, gdy odwołujesz się do widoku, MySQL wykonuje to zapytanie i zwraca wynik.

Zastosowanie

Widoki są używane w różnych scenariuszach:

- **Uproszczenie zapytań**: Zamiast pisać skomplikowane JOIN-y za każdym razem, użytkownik może odwołać się do widoku.
- **Bezpieczeństwo**: Ograniczają dostęp do wrażliwych danych – np. widok pokazuje tylko wybrane kolumny, ukrywając resztę tabeli.
- **Abstrakcja danych**: Ukrywają strukturę bazy danych przed użytkownikami końcowymi, np. w aplikacjach webowych lub raportach.
- **Integracja systemów**: Ułatwiają migrację lub integrację z innymi narzędziami, prezentując dane w spójny sposób.
- **Optymalizacja**: W niektórych przypadkach (z materializowanymi widokami w nowszych wersjach MySQL lub InnoDB) mogą cache'ować wyniki dla szybszego dostępu.

Znaczenie

Widoki mają kluczowe znaczenie w zarządzaniu bazami danych, **ponieważ promują zasadę DRY (Don't Repeat Yourself)** – unikają duplikowania kodu SQL. Poprawiają czytelność i utrzymywalność kodu, ułatwiają kontrolę dostępu (np. via GRANT na widokach) i wspierają modularność w dużych systemach. W

kontekście MySQL, widoki są standardową funkcją od wersji 5.0, co czyni je istotnym narzędziem w projektowaniu skalowalnych aplikacji bazodanowych. Pomagają też w compliance z regulacjami jak GDPR, ograniczając ekspozycję danych osobowych.

Zalety

- **Bezpieczeństwo i kontrola dostępu:** Można nadawać uprawnienia tylko do widoku, bez dostępu do tabel bazowych.
- **Uproszczenie:** Redukują złożoność zapytań dla użytkowników i deweloperów.
- **Aktualność danych:** Widoki zawsze pokazują bieżące dane, bez potrzeby ręcznego odświeżania.
- **Efektywność:** W dużych bazach ułatwiają optymalizację, np. poprzez indeksy na widokach (w MySQL 8.0+).
- **Łatwość utrzymania:** Zmiana w widoku wpływa na wszystkie zapytania go używające, bez modyfikacji kodu aplikacji.

Wady

- **Wydażność:** Ponieważ dane są obliczane dynamicznie, skomplikowane widoki mogą spowalniać zapytania, zwłaszcza przy dużych zbiorach danych (brak materializacji w standardowych widokach MySQL).
- **Brak modyfikacji:** Widoki są tylko do odczytu (nie można INSERT/UPDATE/DELETE bezpośrednio, chyba że przez wyzwalacze lub updatable views w prostych przypadkach).
- **Zależności:** Zmiana struktury tabel bazowych może złamać widok, wymagając jego odtworzenia.
- **Ograniczona funkcjonalność:** Nie obsługują wszystkich operacji (np. ORDER BY w definicji widoku bez TOP/LIMIT), a w MySQL nie ma natywnej materializacji (trzeba używać tabel tymczasowych lub zewnętrznych narzędzi).
- **Złożoność debugowania:** Błędy w widokach mogą być trudne do namierzenia, bo są "czarną skrzynką" dla użytkownika.

Tworzenie widoku

```
CREATE VIEW aktywni_klienci AS
SELECT k.imie, k.nazwisko, k.email, z.data, z.kwota
FROM klienci k
JOIN zamowienia z ON k.id_klienta = z.id_klienta
WHERE z.data > DATE_SUB(NOW(), INTERVAL 1 YEAR);
```

Użycie widoku

```
SELECT * FROM aktywni_klienci;
```

Tabele tymczasowe (ang. *temporary tables*) w MySQL to specjalne tabele, które **istnieją tylko w ramach bieżącej sesji połączenia z bazą danych**. Jest ona automatycznie usuwana po zakończeniu sesji (np. po rozłączeniu się z serwerem MySQL). Tabele tymczasowe są **przydatne do przechowywania pośrednich wyników zapytań, przetwarzania danych tymczasowo lub unikania konfliktów z trwałymi tabelami. Są widoczne tylko dla użytkownika, który je utworzył, i nie wpływają na inne sesje.**

```
-- Tworzenie tabeli tymczasowej
CREATE TEMPORARY TABLE temp_suma_zamowien AS
SELECT id_klienta, SUM(kwota) AS calkowita_kwota
FROM zamowienia
GROUP BY id_klienta;

-- Użycie tabeli tymczasowej
SELECT * FROM temp_suma_zamowien WHERE calkowita_kwota > 200;

-- Usunięcie (opcjonalne, bo zniknie po sesji)
DROP TEMPORARY TABLE IF EXISTS temp_suma_zamowien;
```

Zastosowanie

Tabele tymczasowe są używane w scenariuszach, gdzie potrzebne jest tymczasowe przechowywanie pośrednich wyników:

- **Przetwarzanie złożonych zapytań:** Podzielenie skomplikowanego query na etapy, np. agregacja danych przed JOIN-em.
- **Optymalizacja wydajności:** Przechowywanie wyników podzapytań, aby uniknąć wielokrotnego obliczania (np. w raportach).
- **Testowanie i debugowanie:** Tworzenie mock danych bez modyfikacji stałych tabel.
- **Procedury przechowywane i wyzwalacze:** Przechowywanie tymczasowych danych w skryptach SQL.
- **Integracja z aplikacjami:** W sesjach użytkownika, np. w web app, do personalizowanych obliczeń bez obciążania głównej bazy.

Znaczenie

Tabele tymczasowe mają istotne znaczenie w optymalizacji i zarządzaniu zasobami w MySQL, szczególnie w środowiskach o wysokim obciążeniu. Pozwalają na efektywne zarządzanie pamięcią (mogą być w pamięci lub na dysku, w zależności od silnika jak InnoDB lub MEMORY), co poprawia wydajność zapytań. W kontekście MySQL (od wersji 3.23, ale ulepszone w nowszych), wspierają skalowalność aplikacji, zapobiegając blokadom i konfliktom w wielosesyjnych środowiskach. Są kluczowe w compliance, bo nie pozostawiają śladów po sesji, co pomaga w ochronie danych tymczasowych.

Zalety

- **Izolacja sesji:** Widoczne tylko dla bieżącego połączenia, co zapobiega konfliktom w aplikacjach wieloużytkownikowych.
- **Automatyczne czyszczenie:** Usuwane po zakończeniu sesji, co oszczędza miejsce i upraszcza zarządzanie.
- **Wydajność:** Mogą być szybsze niż podzapytania, zwłaszcza przy dużych zbiorach danych (np. używając ENGINE=MEMORY dla tabel w RAM).
- **Elastyczność:** Wspierają indeksy, klucze obce i większość operacji jak zwykłe tabele.
- **Bezpieczeństwo:** Nie wpływają na trwałe dane, idealne do testów lub tymczasowych obliczeń.

Wady

- **Ograniczony zakres:** Niewidoczne poza sesją, co uniemożliwia udostępnianie danych między połączeniami.

- **Zużycie zasobów:** Przy dużych tabelach mogą zużywać dużo pamięci/dysku, prowadząc do błędów (np. "Out of memory").
- **Brak trwałości:** Dane giną po awarii sesji lub serwera, co wymaga ponownego tworzenia.
- **Złożoność w transakcjach:** W InnoDB, mogą być rollback'owane, ale nie zawsze zachowują się jak trwałe tabele.
- **Ograniczona funkcjonalność:** Nie można ich replikować w klastrach MySQL, a w starszych wersjach miały problemy z LOCK-ami.

Uwaga: jeśli utworzysz tymczasową tabelę o takiej samej nazwie jak istniejąca stała tabela, MySQL będzie używać wersji tymczasowej w danej sesji. Po jej usunięciu znów zobaczysz oryginalną tabelę.

Indeksy i klucze

Możesz w tabelach tymczasowych: definiować **PRIMARY KEY, UNIQUE, INDEX** itp.

```
CREATE TEMPORARY TABLE produkty_tmp (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nazwa VARCHAR(100),
    cena DECIMAL(10,2),
    INDEX (cena)
);
```

- Jednak nie możesz tworzyć indeksów **FULLTEXT** ani **SPATIAL** w tabelach tymczasowych.

Indeksy są definiowane w celu zwiększenia prędkości wykonywania operacji pobierania danych z tabeli. To uporządkowane struktury zawierające dane z wybranych kolumn tabeli.

Indeksy można budować na etapie tworzenia tabel lub definiować je w już istniejącej tabeli.

Indeksy służą głównie do optymalizacji wydajności zapytań SQL:

- **Przyspieszanie SELECT:** Szybsze wyszukiwanie po warunkach WHERE, JOIN, ORDER BY czy GROUP BY.
- **Unikanie pełnego skanowania tabeli:** Zamiast czytać wszystkie rekordy, MySQL skanuje tylko indeks.
- **Egzekwowanie unikalności:** Niektóre indeksy (np. UNIQUE) zapobiegają duplikatom.
- **Wsparcie dla kluczy obcych:** Automatycznie tworzone dla FOREIGN KEY, aby przyspieszyć sprawdzanie integralności.

Jednak indeksy mają koszt: zajmują dodatkowe miejsce na dysku i spowalniają operacje INSERT/UPDATE/DELETE, bo indeks musi być aktualizowany.

Zastosowanie

Indeksy są stosowane w scenariuszach wymagających częstego wyszukiwania lub sortowania:

- **Bazy danych z dużą ilością danych:** Np. w e-commerce do szybkiego wyszukiwania produktów po cenie czy kategorii.

- **Raporty i analizy:** Przyspieszają agregacje (SUM, COUNT) na dużych tabelach.
- **JOIN-y między tabelami:** Indeksy na kolumnach łączących (np. id_klienta) redukują czas wykonania.
- **Filtry w WHERE:** Dla kolumn często używanych w warunkach, jak data czy status.
- **Optymalizacja zapytań:** Narzędzia jak EXPLAIN pomagają identyfikować, gdzie dodać indeksy.

Zaletą stosowania indeksów jest ograniczenie ilości danych odczytywanych z bazy, przyspieszenie wyszukiwania informacji oraz sortowanie danych.

Wadą jest to, że zajmują na dysku dodatkowe miejsce, muszą być na bieżąco aktualizowane, a każde wstawienie, usunięcie lub zaktualizowanie danych w tabeli wiąże się z aktualizacją wszystkich zdefiniowanych dla niej indeksów.

Przykład zastosowania na tabelach klienci i zamówienia

-- Indeks na id_klienta w zamówieniach (dla szybkich JOIN-ów i WHERE)

```
CREATE INDEX idx_id_klienta ON zamówienia(id_klienta);
```

-- Indeks na data w zamówieniach (dla filtrów po dacie i ORDER BY)

```
CREATE INDEX idx_data ON zamówienia(data);
```

-- Kompozytowy indeks na id_klienta i data (dla zapytań z oboma warunkami)

```
CREATE INDEX idx_klient_data ON zamówienia(id_klienta, data);
```

-- Unikalny indeks na email w klientach (zapobiega duplikatom)

```
CREATE UNIQUE INDEX idx_email ON klienci(email);
```

Przykład testu efektywności indeksu:

1. Wykonaj zapytanie testowe:

```
SELECT * FROM zamówienia WHERE data > '2024-01-05';
```

2. Wykonaj EXPLAIN przed dodaniem indeksu:

```
EXPLAIN SELECT * FROM zamówienia WHERE data > '2024-01-05';
```

Oczekiwany wynik (przybliżony, bez indeksu):

- id: 1

- select_type: SIMPLE
- table: zamowienia
- type: **ALL** (pełny skan tabeli – wolne, sprawdza wszystkie wiersze)
- possible_keys: NULL
- key: NULL
- key_len: NULL
- rows: ~3 (lub więcej, cała tabela)
- Extra: Using where

To oznacza, że MySQL musi przejrzeć każdy wiersz w tabeli.

3. Dodanie indeksu na data

CREATE INDEX idx_data **ON** zamowienia(data);

4. EXPLAIN po dodaniu indeksu:

EXPLAIN SELECT * FROM zamowienia **WHERE** data > '2024-01-05';

Oczekiwany wynik (przybliżony, z indeksem):

- id: 1
- select_type: SIMPLE
- table: zamowienia
- type: **range** (użycie indeksu dla zakresu – szybkie)
- possible_keys: idx_data
- key: idx_data
- rows: ~5 (szacowana liczba pasujących wierszy, nie cała tabela)
- Extra: Using index condition

Teraz MySQL używa struktury B-tree indeksu, aby szybko znaleźć rekordy w zakresie, bez skanowania całej tabeli.

Różnica w praktyce

- **Przed:** Pełny skan (type: ALL) – czas rośnie liniowo z rozmiarem tabeli (wolne na dużych danych).
- **Po:** Skan zakresu (type: range) – czas logarytmiczny, znacznie szybszy.
- **Kiedy pomaga:** W dużych tabelach (tysiące+ rekordów) różnica w czasie wykonania może być od sekund do milisekund. Użyj **ANALYZE TABLE** zamowienia; po wstawieniu danych, aby zaktualizować statystyki.

