

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()	Wydrebnia 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: ERD (Diagram związków encji ang. Entity Relationship Diagram). Właściwości kolumn (pól) w MySQL: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, DEFAULT, CHECK, AUTO_INCREMENT, ENUM, COMMENT. Polecenie DELETE i DROP

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.

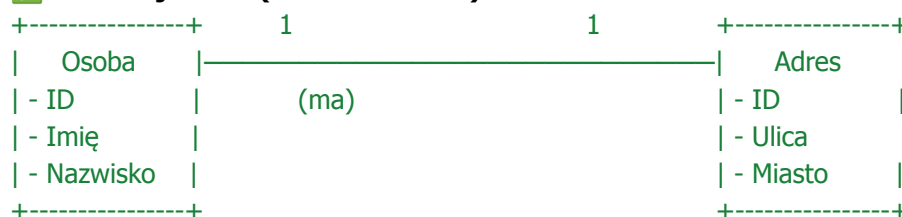
2) Relacja 1:N (One to Many)

Jeden klient może mieć wiele zamówień.

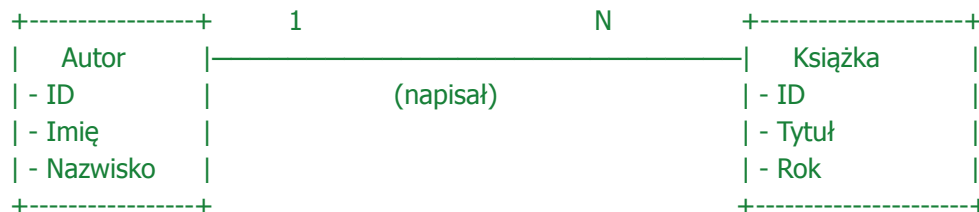
3) Relacja N:M (Many to Many)

Tworzy się tabelę pośredniczącą.

✓ 1. Relacja 1 : 1 (Osoba — Adres)

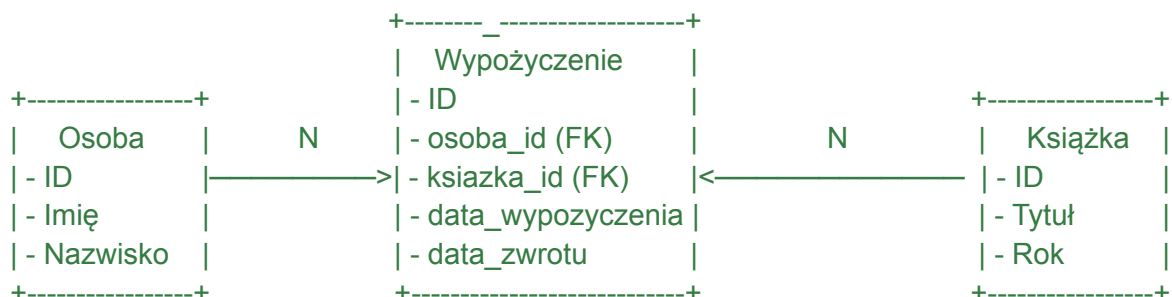


✓ 2. Relacja 1 : N (Autor — Książka)



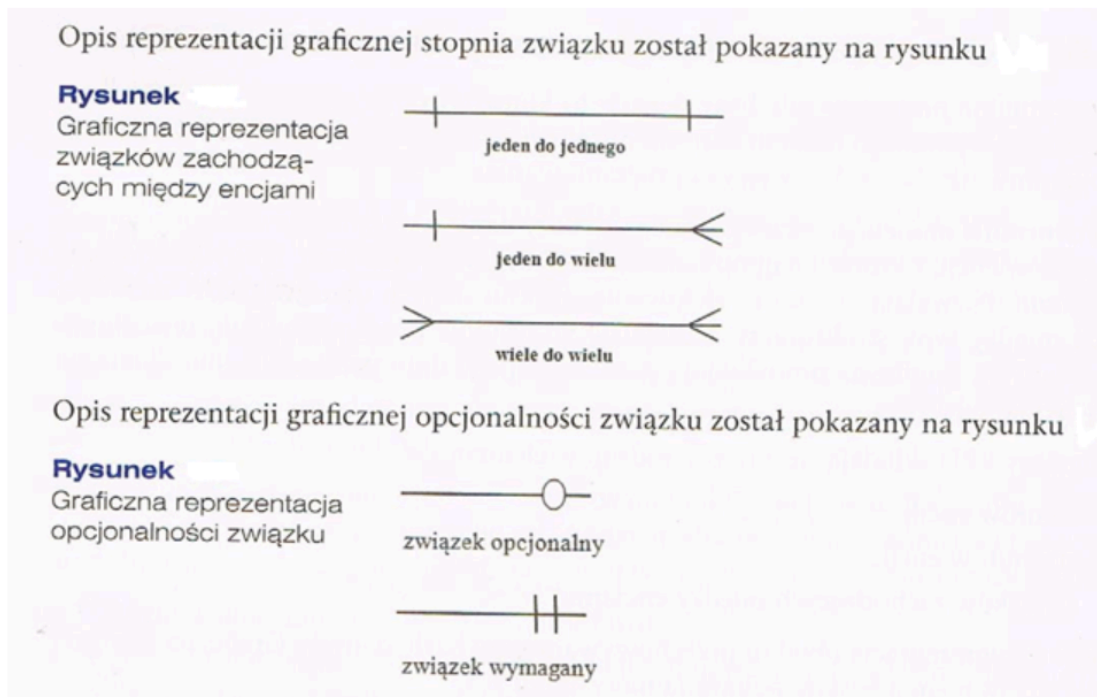
✓ 3. Relacja N : N (Osoba — Książka) przez tabelę Wypożyczenie

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



N : N

(wiele osób wypożycza wiele książek)



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

Właściwości kolumn (pól) w MySQL: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, DEFAULT, CHECK, AUTO_INCREMENT, ENUM, COMMENT

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:

- imię 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)

Dodanie UNIQUE do istniejącej tabeli

```
ALTER TABLE klienci  
ADD UNIQUE (email);
```

Różnica: PRIMARY KEY vs UNIQUE

Cecha	PRIMARY KEY	UNIQUE
Musi być unikalne	✓ Tak	✓ Tak
Może być NULL	✗ Nie	✓ Tak
Można mieć więcej niż jeden?	✗ Nie (tylko jeden PK na tabelę)	✓ Tak (wiele UNIQUE)
Tworzy indeks	✓ Tak	✓ Tak

✓ 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 artykuly (  
  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.

Polecenie **DELETE** i **DROP**

♦ **DELETE FROM**

Polecenie:

```
DELETE FROM nazwa_tabeli;
```

Usuwa rekordy (wiersze) z tabeli, ale:

- **nie usuwa struktury tabeli**, kolumn ani jej definicji,
- **nie resetuje auto_increment** (chyba że użyjesz TRUNCATE),
- może usuwać pojedyncze wiersze lub wszystkie — zależnie od warunku WHERE.

Przykłady:

Usuń wszystkie rekordy:

```
DELETE FROM users;
```

Usuń tylko wybrane:

```
DELETE FROM users WHERE id = 5;
```

♦ DROP

Polecenie:

DROP TABLE nazwa_tabeli;

Usuwa całą tabelę z bazy danych, czyli:

- usuwa wszystkie dane,
- usuwa strukturę tabeli (kolumny, indeksy, klucze),
- usuwa definicję tabeli z katalogu bazy.

Po wykonaniu DROP tabela **przestaje istnieć**.

Przykłady:

Usuń tabelę:

DROP TABLE users;

Usuń całą bazę danych:

DROP DATABASE sklep;

Lekcja

Temat: Replair. Akronym ACID, kategorie poleceń w SQL. System Zarządzania Bazą Danych (DBMS – DataBase Management System)

♦ Polecenie **REPAIR TABLE** w MySQL służy do **naprawy uszkodzonych tabel** oraz do **optymalizacji** pewnych typów tabel. Działa jednak tylko dla wybranych silników — głównie **MyISAM** oraz **ARCHIVE**.

Jeśli tabela MyISAM została uszkodzona (np. po awarii serwera), **REPAIR TABLE** próbuje:

- odbudować indeksy,
- odtworzyć strukturę danych,
- odzyskać jak najwięcej wierszy.

Składnia

REPAIR TABLE nazwa_tabeli;

Dodatkowe opcje:

- **QUICK** – naprawia tylko indeksy, bez skanowania danych
- **EXTENDED** – dogłębna naprawa, rekonstruuje plik danych (najwolniejsza)
- **USE_FRM** – odbudowuje indeksy na podstawie pliku .frm (tylko MyISAM)

REPAIR TABLE *nie naprawia* tabel InnoDB.

♦ **Akrónim ACID w SQL oznacza cztery kluczowe właściwości transakcji w systemach baz danych:**

A – Atomicity (Atomowość)

Transakcja jest niepodzielna: albo wykonuje się w całości, albo wcale.

C – Consistency (Spójność)

Transakcja musi pozostawić bazę danych w stanie zgodnym z regułami i ograniczeniami (constraints).

I – Isolation (Izolacja)

Równocześnie wykonywane transakcje nie powinny wzajemnie sobie przeszkadzać — każda działa tak, jakby była wykonywana osobno.

D – Durability (Trwałość)

Po zatwierdzeniu transakcji (COMMIT) jej skutki są trwałe i nie zostaną utracone, nawet w przypadku awarii.

♦ **Podstawowe kategorie poleceń w SQL to:**

- **DDL (Data Definition Language)** – definiowanie struktury bazy danych (np. tworzenie tabel).
- **DML (Data Manipulation Language)** – manipulacja danymi (np. wstawianie, aktualizacja, usuwanie).
- **DCL (Data Control Language)** – zarządzanie uprawnieniami (np. GRANT, REVOKE).
- **DQL (Data Query Language)** – pobieranie danych (np. SELECT).
- **TCL (Transaction Control Language)** – zarządzanie transakcjami (np. COMMIT, ROLLBACK).

♦ **System Zarządzania Bazą Danych (DBMS – DataBase Management System)**

to oprogramowanie, które umożliwia:

- tworzenie baz danych,
- zapisywanie, modyfikowanie i usuwanie danych,
- zarządzanie dostępem użytkowników,
- zapewnianie bezpieczeństwa i integralności danych,
- wykonywanie zapytań (np. SQL),
- jednoczesny dostęp wielu użytkowników.

Prościej:

👉 **DBMS to program do zarządzania danymi w bazie – np. MySQL, PostgreSQL, Oracle, SQL Server.**

✓ Jakie mechanizmy są NIEZBĘDNE dla Systemu Zarządzania Bazą Danych?

Wszystkie SZBD muszą mieć pewne podstawowe mechanizmy — zwykle wymienia się:

1. Mechanizm składowania danych

Przechowywanie danych na dysku, w tabelach, indeksach itp.

2. Mechanizm dostępu do danych / język zapytań (np. SQL)

Możliwość pobierania, wstawiania, usuwania, aktualizowania danych.

3. Mechanizmy bezpieczeństwa

- **autoryzacja i autentykacja,**
- **role, użytkownicy,**
- **uprawnienia.**

4. Mechanizmy kontroli współbieżności (concurrency control)

Zapewniają poprawną pracę wielu użytkowników *jednocześnie*.

5. Mechanizmy zapewnienia integralności danych

- **klucze główne,**
- **klucze obce,**
- **ograniczenia (NOT NULL, UNIQUE, CHECK).**

Chronią przed niepoprawnymi danymi.

6. Mechanizmy odtwarzania po awarii (recovery)

Przywracają działanie po:

- **awarii systemu,**
- **utracie zasilania,**
- **błędach sprzętu.**

Zapisywanie logów transakcyjnych, backupy itp.

7. Mechanizmy zarządzania transakcjami (ACID)

Każdy SZBD musi obsługiwać transakcje zgodnie z zasadą:

- **A atomicity – niepodzielność**
- **C consistency – spójność**
- **I isolation – izolacja**
- **D durability – trwałość**

To fundament poprawnej pracy.

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.

```
-- 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

- **Wydajność:** 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.

Lekcja

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

```
-- 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

- **Wydajność**: 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.

Lekcja

Temat: INDEKS

Indeksy w MySQL to struktury danych, które przyspieszają wyszukiwanie i sortowanie danych w tabelach bazy danych. Bez indeksu, MySQL musiałby skanować całą tabelę (tzw. full table scan), co jest nieefektywne dla dużych zbiorów danych. Indeks działa jak spis treści w książce – pozwala szybko znaleźć konkretne wiersze bez sprawdzania wszystkich.

Podstawowe zasady działania:

- **Przyspieszanie zapytań:** Indeksy są używane w klauzulach
 - WHERE, JOIN, ORDER BY i GROUP BY

Na przykład, jeśli masz indeks na kolumnie **email**, następnie wykonasz zapytanie **SELECT * FROM users WHERE email = 'example@domain.com'** uzyskanie wyniku będzie szybkie, bo MySQL użyje indeksu do bezpośredniego dostępu do wierszy.

- **Koszt:** Indeksy zużywają miejsce na dysku i spowalniają operacje
 - INSERT, UPDATE i DELETE,

bo po każdej zmianie indeks musi być aktualizowany.

Typy indeksów w MySQL

MySQL obsługuje kilka typów indeksów, w zależności od silnika bazy danych (np. InnoDB, MyISAM). Poniżej wymieniam główne typy wraz z krótkim opisem. Typy te definiuje się podczas tworzenia indeksu za pomocą słów kluczowych w SQL.

1. **PRIMARY KEY:**

- Unikalny indeks, który służy jako główny klucz identyfikujący wiersze w tabeli.
- **Nie pozwala na wartości NULL i musi być unikalny.**
- W InnoDB jest to clustered index – dane tabeli są fizycznie posortowane według tego klucza.
- Przykład: PRIMARY KEY (id).

2. **UNIQUE:**

- **Zapewnia unikalność wartości w kolumnie lub grupie kolumn.**
- **Mogą być wartości NULL** (w zależności od definicji), **ale wartości nie mogą się powtarzać.**
- Używany do egzekwowania integralności danych.
- Przykład: UNIQUE INDEX idx_email (email).

3. **INDEX** (lub KEY, zwykły indeks):

- Podstawowy typ indeksu, **nie wymuszający unikalności.**
- Przyspiesza wyszukiwanie, sortowanie i łączenie tabel.
- **Może być na jednej lub wielu kolumnach (composite index).**
- Przykład: INDEX idx_last_name (last_name).

4. **FULLTEXT:**

- Specjalny **indeks do wyszukiwania pełnotekstowego w kolumnach tekstowych** (np. VARCHAR, TEXT).
- Obsługuje wyszukiwanie słów kluczowych, z obsługą stop words, stemmingu i rankingiem relevancji.
- Dostępny w InnoDB i MyISAM.
- Przykład: FULLTEXT INDEX idx_content (content).

5. **SPATIAL:**

- **Indeks dla danych przestrzennych** (geometria, punkty, linie itp.).

- Używa struktury R-tree do efektywnego wyszukiwania przestrzennego (np. bliskość, zawieranie).
- Wymaga kolumn typu GEOMETRY.
- Przykład: `SPATIAL INDEX idx_location (location).`

6. HASH:

- **Indeks oparty na hashowaniu, szybki dla równości, ale nie dla zakresów.**
- Dostępny głównie w silniku MEMORY (HEAP), lub implicitnie w InnoDB dla niektórych operacji.
- Nie jest powszechnie używany w InnoDB, gdzie dominuje B-tree.
- Przykład: `INDEX idx_hash USING HASH (column).`

```
DROP TABLE IF EXISTS example_table;
```

```
CREATE TABLE example_table (
  id INT,
  name VARCHAR(255),
  email VARCHAR(255),
  description TEXT,
  location GEOMETRY NOT NULL,
  code CHAR(10)
);
```

1. PRIMARY KEY

- **Tworzenie** (podczas tworzenia tabeli lub dodawanie później):

-- Podczas tworzenia tabeli

```
CREATE TABLE example_table (
  id INT AUTO_INCREMENT PRIMARY KEY
);
```

-- Dodawanie później (jeśli nie istnieje)

```
ALTER TABLE example_table ADD PRIMARY KEY (id);
```

- **Edycja** (zmiana kolumny PRIMARY KEY wymaga usunięcia i dodania nowego; nie można bezpośrednio edytować):

-- Usuń istniejący PRIMARY KEY (jeśli to możliwe, np. nie AUTO_INCREMENT)

```
ALTER TABLE example_table DROP PRIMARY KEY;
```

-- Dodaj nowy na innej kolumnie (np. email, ale musi być unikalne i nie NULL)

```
ALTER TABLE example_table ADD PRIMARY KEY (email);
```

Uwaga: Jeśli PRIMARY KEY jest AUTO_INCREMENT, edycja może wymagać rekonstrukcji tabeli.

- **Usuwanie:**

```
ALTER TABLE example_table DROP PRIMARY KEY;
```

Uwaga: Tabela może istnieć bez PRIMARY KEY, ale to niezalecane w InnoDB.

2. UNIQUE

- **Tworzenie:**

-- Podczas tworzenia tabeli

```
CREATE TABLE example_table (
  email VARCHAR(255) UNIQUE
);
```

-- Dodawanie później

```
ALTER TABLE example_table ADD UNIQUE INDEX idx_unique_email (email);
```

Lub:

```
CREATE UNIQUE INDEX indeks_unique_email ON example_table (email);
```

- **Edycja** (nie bezpośrednia; usuń i dodaj nowy, np. zmień kolumny):

-- Usuń istniejący

```
ALTER TABLE example_table DROP INDEX indeks_unique_email ;
```

-- Dodaj zmodyfikowany (np. na dwóch kolumnach)

```
ALTER TABLE example_table ADD UNIQUE INDEX idx_unique_name_email (name, email);
```

- **Usuwanie:**

```
ALTER TABLE example_table DROP INDEX idx_unique_email;
```

Lub:

```
DROP INDEX idx_unique_email ON example_table;
```

3. INDEX (lub KEY, zwykły indeks)

- **Tworzenie:**

-- Podczas tworzenia tabeli

```
CREATE TABLE example_table (  
    name VARCHAR(255),  
    INDEX idx_name (name)  
);
```

-- Dodawanie później

```
ALTER TABLE example_table ADD INDEX idx_name (name);
```

Lub:

```
CREATE INDEX idx_name ON example_table (name);
```

Dla kompozytowego:

```
CREATE INDEX idx_name ON example_table (name, email)
```

- **Edycja** (nie bezpośrednia; usuń i dodaj nowy, np. dodaj kolumnę):

-- Usuń istniejący

```
ALTER TABLE example_table DROP INDEX idx_name;
```

-- Dodaj zmodyfikowany (np. kompozytowy)

```
ALTER TABLE example_table ADD INDEX idx_name_email (name, email);
```

Można zmienić nazwę:

```
ALTER TABLE example_table RENAME INDEX idx_name TO idx_new_name;
```

- **Usuwanie:**

```
ALTER TABLE example_table DROP INDEX idx_name;
```

Lub:

```
DROP INDEX idx_name ON example_table;
```

4. FULLTEXT

- **Tworzenie:**

-- Podczas tworzenia tabeli

```
CREATE TABLE example_table (  
    description TEXT,
```

```
FULLTEXT INDEX idx_fulltext_desc (description)
);
```

-- Dodawanie później

```
ALTER TABLE example_table ADD FULLTEXT INDEX idx_fulltext_desc (description);
```

Lub:

```
CREATE FULLTEXT INDEX idx_fulltext_desc ON example_table (description);
```

- **Edycja** (nie bezpośrednia; usuń i dodaj nowy, np. zmień kolumny lub parser):

-- Usuń istniejący

```
ALTER TABLE example_table DROP INDEX idx_fulltext_desc;
```

-- Dodaj zmodyfikowany (np. na dwóch kolumnach z parserem)

```
ALTER TABLE example_table ADD FULLTEXT INDEX idx_fulltext_name_desc (name, description);
```

- **Usuwanie:**

```
ALTER TABLE example_table DROP INDEX idx_fulltext_desc;
```

Lub:

```
DROP INDEX idx_fulltext_desc ON example_table;
```

5. SPATIAL

- **Tworzenie:**

-- Podczas tworzenia tabeli (kolumna musi być GEOMETRY lub podobna)

```
CREATE TABLE example_table (
    location GEOMETRY NOT NULL,
    SPATIAL INDEX idx_spatial_loc (location)
);
```

-- Dodawanie później

```
ALTER TABLE example_table ADD SPATIAL INDEX idx_spatial_loc (location);
```

Lub:

```
CREATE SPATIAL INDEX idx_spatial_loc ON example_table (location);
```

- **Edycja** (nie bezpośrednia; usuń i dodaj nowy):

-- Usuń istniejący

```
ALTER TABLE example_table DROP INDEX idx_spatial_loc;
```

-- Dodaj zmodyfikowany (np. jeśli zmienisz definicję kolumny)

```
ALTER TABLE example_table ADD SPATIAL INDEX idx_spatial_new (location);
```

Uwaga: SPATIAL wymaga kolumny NOT NULL w nowszych wersjach.

- **Usuwanie:**

```
ALTER TABLE example_table DROP INDEX idx_spatial_loc;
```

Lub:

```
DROP INDEX idx_spatial_loc ON example_table;
```

6. HASH

- **Tworzenie:**

-- Podczas tworzenia tabeli (najlepiej w silniku MEMORY lub Aria)

```
CREATE TABLE example_table (
    code CHAR(10),
    INDEX idx_hash_code USING HASH (code)
) ENGINE=MEMORY;
```

-- Dodawanie później

```
ALTER TABLE example_table ADD INDEX idx_hash_code USING HASH (code);
```

Lub:

```
CREATE INDEX idx_hash_code ON example_table (code) USING HASH;
```

Uwaga: W InnoDB HASH jest emulowany przez B-tree w większości przypadków.

- **Edycja** (nie bezpośrednia; usuń i dodaj nowy, np. zmień na B-tree):

-- Usuń istniejący

```
ALTER TABLE example_table DROP INDEX idx_hash_code;
```

-- Dodaj zmodyfikowany (np. bez HASH, czyli domyślny B-tree)

```
ALTER TABLE example_table ADD INDEX idx_new_code (code);
```

- **Usuwanie:**

```
ALTER TABLE example_table DROP INDEX idx_hash_code;
```

Lub:

```
DROP INDEX idx_hash_code ON example_table;
```

Ogólne uwagi:

- Po operacjach sprawdź indeksy: `SHOW INDEX FROM example_table;`

Indeksy w MySQL (dla silnika InnoDB, który jest domyślny) są zazwyczaj oparte na strukturze **B-tree** (Balanced Tree), co zapewnia zrównoważone drzewo wyszukiwania binarnego.

B-tree (Balanced Tree) to zrównoważona struktura drzewiasta używana w systemach baz danych (jak MySQL) do indeksowania danych, aby efektywnie obsługiwać operacje na dużych zbiorach danych przechowywanych na dyskach.

Poniżej wyjaśnię strukturę krok po kroku. Zakładam B-tree **rzędu m** (gdzie m to maksymalna liczba dzieci węzła, zwana też stopniem drzewa). W praktyce m zależy od rozmiaru strony dyskowej (np. w MySQL strona indeksu to często 16 KB).

1. Podstawowe właściwości B-tree

- **Zrównoważenie:** Wszystkie liście (węzły bez dzieci) są na tym samym poziomie głębokości. To zapobiega degeneracji drzewa w liniową strukturę (jak w niezrównoważonych drzewach binarnych).
- **Liczba kluczy w węzłach:**
 - Każdy węzeł (oprócz korzenia) ma co najmniej $\lceil m/2 \rceil - 1$ kluczy (minimalna liczba, aby uniknąć underflow).
 - Maksymalna liczba kluczy w węźle to **m - 1**.
 - Korzeń może mieć od 1 do m-1 kluczy (może być mniejszy).
- **Liczba dzieci:** Jeśli węzeł ma k kluczy, to ma dokładnie **k + 1** dzieci (dla węzłów wewnętrznych).
- **Klucze posortowane:** W każdym węźle klucze są posortowane rosnąco. Wartości mniejsze niż klucz i-tego idą do i-tego dziecka, większe – do (i+1)-tego.

2. Typy węzłów

B-tree składa się z trzech rodzajów węzłów:

- **Korzeń (Root):** Najwyższy węzeł, może być liściem lub mieć dzieci. Jeśli drzewo ma tylko jeden węzeł, to jest nim korzeń.
- **Węzły wewnętrzne (Internal Nodes):** Zawierają klucze i wskaźniki do dzieci. Nie przechowują danych (w czystym B-tree dane mogą być w dowolnych węzłach, ale w wariantach jak B+tree – tylko w liściach).

- **Węzły liściowe (Leaf Nodes):** Zawierają klucze i dane (lub wskaźniki do danych). Nie mają dzieci.

Kiedy stosować B-tree

Stosuj B-tree, gdy:

- **Kolumna jest często używana w warunkach wyszukiwania, sortowania lub łączenia tabel.**
- **Zapytania obejmują równość, zakresy lub prefix matching.**
- **Dane są dynamiczne (częste INSERT/UPDATE/DELETE),** bo B-tree dobrze radzi sobie z rebalansowaniem.
- **Tabela jest duża** – B-tree redukuje full table scan do logarytmicznego czasu.
- **Dla composite indexes (wielokolumnowych), gdzie kolejność kolumn ma znaczenie (lewy prefix rule).**

B-tree jest szczególnie efektywny, gdy selektywność jest wysoka (mało duplikatów w kolumnie), a indeks pokrywa zapytanie (covering index – bez potrzeby odczytu całej tabeli).

Przykłady zapytań, gdzie B-tree pomaga

Oto typowe scenariusze z przykładami SQL. Zakładam tabelę `users` z indeksem B-tree na `age (INT)` i composite na `(last_name, first_name)`.

1. **Równość (=):**
 - Zapytanie: `SELECT * FROM users WHERE age = 30;`
 - Dlaczego B-tree: Szybko lokalizuje dokładne dopasowania.
2. **Zakresy (>, <, BETWEEN, >=, <=):**
 - Zapytanie: `SELECT * FROM users WHERE age BETWEEN 20 AND 40;`
 - Dlaczego B-tree: Efektywne skanowanie zakresowe dzięki połączonym liściom (w B+tree).
3. **Sortowanie (ORDER BY):**
 - Zapytanie: `SELECT * FROM users ORDER BY last_name ASC LIMIT 10;`
 - Dlaczego B-tree: Dane są już posortowane w indeksie, unika dodatkowego sortu.
4. **Grupowanie (GROUP BY):**
 - Zapytanie: `SELECT last_name, COUNT(*) FROM users GROUP BY last_name;`
 - Dlaczego B-tree: Przyspiesza agregację po indeksowanej kolumnie.
5. **Łączenie tabel (JOIN):**
 - Zapytanie: `SELECT u.name, o.order_date FROM users u JOIN orders o ON u.id = o.user_id WHERE u.age > 25;`
 - Dlaczego B-tree: Szybkie matching po kluczu obcym.
6. **LIKE z prefixem (bez wildcarda na początku):**
 - Zapytanie: `SELECT * FROM users WHERE last_name LIKE 'Smith%';`
 - Dlaczego B-tree: Traktuje jako zakres (od 'Smith' do 'Smithz...').

Sprawdź użycie indeksu za pomocą EXPLAIN SELECT ... – jeśli pokazuje "Using index", to B-tree działa.

Kiedy nie stosować B-tree

Nie stosuj B-tree (lub nie jest on efektywny), gdy:

- Zapytanie nie korzysta z indeksu (np. niska selektywność – kolumna z wieloma duplikatami, jak płeć: 'M'/'F').
- Indeks spowalnia modyfikacje (zbyt wiele indeksów na tabeli – każdy UPDATE wymaga aktualizacji wszystkich).
- Dla specjalnych typów danych lub zapytań, gdzie lepsze są inne struktury (HASH, FULLTEXT, SPATIAL).
- Optimizer wybiera full table scan, bo tabela jest mała lub indeks nie jest selektywny.

Przykłady zapytań, gdzie B-tree **nie pomaga lub nie jest używany**:

1. **LIKE z wildcardem na początku:**

- Zapytanie: `SELECT * FROM users WHERE last_name LIKE '%mith';`
- Dlaczego nie: Nie może użyć zakresu – wymaga full scan.

2. **Funkcje na kolumnie:**

- Zapytanie: `SELECT * FROM users WHERE UPPER(last_name) = 'SMITH';`
- Dlaczego nie: Funkcja (UPPER) uniemożliwia użycie indeksu (chyba że użyjesz functional index w nowszych wersjach).

3. **Nierówności z OR (bez optymalizacji):**

- Zapytanie: `SELECT * FROM users WHERE age < 20 OR age > 60;`
- Dlaczego nie: Może wymagać dwóch skanów lub full scan, jeśli nie ma UNION.

4. **Wyszukiwanie pełnotekstowe:**

- Zapytanie: `SELECT * FROM articles WHERE MATCH(content) AGAINST('keyword');`
- Dlaczego nie: Użyj FULLTEXT index (inverted index, nie B-tree).

5. **Dane przestrzenne:**

- Zapytanie: `SELECT * FROM locations WHERE ST_Contains(geom, POINT(1,2));`
- Dlaczego nie: Użyj SPATIAL index (R-tree).

6. **Tylko równość w małych tabelach (MEMORY engine):**

- Zapytanie: `SELECT * FROM temp_table WHERE code = 'ABC';`
- Dlaczego nie: Lepszy HASH index dla czystej równości (szybszy, ale nie dla zakresów).