Czym jest baza danych

Cztery pojęcia, które się myli, model relacyjny, klucze, relacje i mapa rodzin silników. Rozdział, na którym opiera się cała reszta półki.

- Bazy danych: od kart perforowanych do SQL
- Jak korzystać z tej książki
- Baza danych, silnik, instancja, schemat: cztery pojęcia, które się myli
- Model relacyjny: tabela, wiersz, kolumna, dziedzina
- Klucze: główny, obcy, unikalny, naturalny, sztuczny
- Relacje: 1:1, 1:N, N:M i tabela łącząca
- Katalog systemowy: skąd baza wie, co w sobie ma
- Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin

Bazy danych: od kart perforowanych do SQL

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

Ta strona otwiera książkę „Bazy danych: fundamenty": skąd wziął się pomysł trzymania danych w tabelach, dlaczego dziś mamy kilka rywalizujących silników zamiast jednego standardu i co czeka na kolejnych stronach.

Skróty użyte na tej stronie (8)

Skrót	Co znaczy
ACM	AWS Certificate Manager: usługa wydająca i odnawiająca certyfikaty TLS dla równoważników, CloudFront i API Gateway
IBM	International Business Machines: firma, w której laboratorium powstał model relacyjny i pierwszy działający silnik SQL
IT	Information Technology: dział albo praca związana z komputerami i infrastrukturą firmy
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
OS/2	system operacyjny firmy IBM i Microsoft z lat 80., z którego pomysłów wyrósł początkowo Windows NT
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
T-SQL	Transact-SQL: dialekt SQL używany w Microsoft SQL Server, wywodzący się z języka zapytań oryginalnego silnika Sybase
WMS	Warehouse Management System: system zarządzania magazynem

Skąd się wzięło

> **Pojęcie: baza danych.** Uporządkowany zbiór danych trzymany tak, żeby dało się w nim szybko znaleźć, zmienić i bezpiecznie zapisać pojedynczą informację, bez czytania całego zbioru od początku do końca.

Zanim powstały pierwsze bazy danych w dzisiejszym rozumieniu, dane firm leżały w plikach ułożonych w kolejności, w jakiej ktoś je kiedyś zapisał, a znalezienie jednego rekordu oznaczało przejście przez cały zbiór po kolei. Wczesne systemy z lat sześćdziesiątych wiązały dane sztywnymi wskaźnikami: dostęp do zamówienia klienta wymagał znajomości dokładnej ścieżki, ustalonej z góry przez projektanta. Zmiana tej struktury oznaczała przepisanie każdego programu, który z niej korzystał.

W czerwcu 1970 roku Edgar Codd, matematyk pracujący w laboratorium IBM w San Jose, opublikował w czasopiśmie Communications of the ACM artykuł opisujący model relacyjny: dane w **tabelach** złożonych z wierszy i kolumn, powiązane wspólnymi wartościami zamiast sztywnych wskaźników. Program pytający o dane mógł od tej pory powiedzieć, czego chce, a nie krok po kroku, jak do tego dotrzeć. IBM, które sprzedawało wtedy własne bazy hierarchiczne, nie od razu wdrożyło pomysł własnego pracownika.

> **Pojęcie: silnik bazy danych.** Program, który zarządza zapisem danych na dysku, pilnuje ich spójności i wykonuje zapytania, jakie do niego trafiają. To on, nie sama baza, decyduje, jak fizycznie poukładać dane, żeby odczyt był szybki.

Dopiero w połowie lat siedemdziesiątych zespół badawczy IBM zbudował System R, pierwszy działający silnik relacyjny, razem z językiem zapytań SEQUEL, skróconym później do SQL. **Pojęcie: zapytanie.** Polecenie w języku SQL, które opisuje, jakie dane mają wrócić z bazy, na przykład `SELECT`, bez podawania silnikowi trasy dotarcia do nich. IBM opublikował wyniki tych prac otwarcie, co pozwoliło innym firmom zbudować własne silniki SQL, zanim samo IBM wypuściło produkt komercyjny.

Jedną z tych firm była mała spółka założona w 1977 roku w Kalifornii przez Larry'ego Ellisona, Boba Minera i Eda Oatesa. W 1979 roku wypuścili pierwszy komercyjny silnik relacyjny z obsługą SQL, nazwany od razu wersją drugą, bo klienci niechętnie kupują produkt oznaczony jako pierwsza wersja. Firma przyjęła nazwę swojego produktu na stałe w 1982 roku: Oracle.

Kilka lat później firma Sybase zbudowała własny silnik SQL Server, a w 1989 roku Microsoft wydał, we współpracy z nią, jego wersję dla systemu OS/2. Współpraca zakończyła się w połowie lat dziewięćdziesiątych: Microsoft zatrzymał nazwę i prawa do kodu na Windows, a MSSQL rozwija się od tamtej pory jako osobny produkt, choć ślady wspólnych korzeni wciąż widać w języku T-SQL.

Równolegle na Uniwersytecie Kalifornijskim w Berkeley Michael Stonebraker prowadził od 1986 roku projekt Postgres, następcę swojej wcześniejszej bazy Ingres. Postgres miał własny język zapytań, dopóki w 1994 roku dwóch studentów nie dopisało mu interpretera SQL. W 1996 roku projekt zmienił nazwę na PostgreSQL i pod tą nazwą rozwija się jako wolne oprogramowanie do dziś.

Na przeciwnym biegunie pod względem historii stanął MySQL, pisany od 1994 roku przez szwedzką firmę założoną przez Davida Axmarka, Allana Larssona i fińskiego programistę Michaela Wideniusa. Nazwa łączy imię córki Wideniusa, My, z SQL, a pierwsza wersja trafiła do użytkowników w maju 1995 roku. Firma trafiła później w ręce Sun Microsystems, a od 2010 roku MySQL należy do Oracle.

Nie każde dane pasują jednak do sztywnych tabel z góry ustalonych kolumn, a gdy internet zaczął obsługiwać ruch liczony w milionach użytkowników naraz, pojedynczy silnik relacyjny bywał zbyt sztywny do łatwego rozproszenia na wiele maszyn. Odpowiedzią stały się silniki NoSQL: dokumentowe, kolumnowe i klucz-wartość, świadomie rezygnujące z części gwarancji relacyjnych na rzecz łatwiejszego skalowania. Termin ukuł w 1998 roku Carlo Strozzi dla własnej bazy bez interfejsu SQL, ale do dzisiejszego znaczenia wrócił dopiero w 2009 roku, gdy Johan Oskarsson szukał nazwy dla spotkania poświęconego nowej fali rozproszonych baz nierelacyjnych.

Jak to działa w jednym obrazku

Niezależnie od silnika droga danych wygląda tak samo: aplikacja wysyła zapytanie do silnika, silnik znajduje właściwą bazę, w bazie znajduje właściwą tabelę, a wynik tej pracy leży fizycznie jako plik na dysku serwera.



Droga zapytania od aplikacji przez silnik bazy danych do pliku na dysku, na przykładzie Optimy i MSSQL na sql01 (kliknięcie otwiera pełny rozmiar)

Dlaczego Zielona

> **Z ŻYCIA WZIĘTE**
>
> Ola postawiła PostgreSQL pod swój pierwszy wewnętrzny panel zespołu, bo z tego silnika korzystała w poprzedniej pracy i nikt jej tego wyboru nie kwestionował. Pół roku później ten sam panel trafił na serwer produkcyjny razem z tą samą bazą, bo przepisanie zapytań pod inny silnik nikomu nie wydawało się warte czasu.

W Zielonej działają naraz trzy silniki, i żaden z nich nie został wybrany dla samej różnorodności. Księgowość Optima kupiona razem z licencją wymaga MSSQL, więc na `sql01` stoi baza `CDN_Zielona`, do której nikt poza samą Optimą bezpośrednio nie pisze. System magazynowy WMS od innego dostawcy wymaga z kolei Oracle, stąd `ora01` z bazą `WMSPRD`, obsługującą czterdzieści skanerów logujących się naraz o szóstej rano w Swarzędzu. Aplikacje pisane przez zespół Oli nie mają takiego przymusu, więc trafiają na `pg01` i PostgreSQL: darmowy, dobrze udokumentowany, bez licencji do pilnowania.

Ta różnorodność nie jest błędem administratora, tylko normalnym skutkiem tego, że wybór silnika bazy danych rzadko należy do działu IT. Aplikacja przychodzi razem ze swoim silnikiem, a administrator dostaje w spadku tyle różnych baz, ile firma kupiła różnych systemów. Reszta tej książki uczy jednego wspólnego języka do rozmowy o wszystkich trzech naraz.

Mapa tej książki

Książka „Bazy danych: fundamenty" składa się z ośmiu rozdziałów. Uczy jednego wspólnego słownika i języka SQL, zanim którakolwiek książka per silnik (MSSQL, Oracle, PostgreSQL, MySQL) zajmie się tym, co w danym silniku jest inne.

Rozdział	Na jakie pytanie odpowiada
Bazy danych: od kart perforowanych do SQL	Skąd się wzięły bazy danych i dlaczego mamy dziś kilka silników
1. Czym jest baza danych	Co to jest baza, silnik, instancja, schemat, klucz i relacja
2. SQL: anatomia zapytania	Jak czytać i pisać podstawowe zapytanie SELECT
3. Łączenie zbiorów	Jak połączyć dane z kilku tabel w jeden wynik (JOIN, UNION, podzapytania)
4. Zmiana danych i struktury	Jak wstawiać, zmieniać i usuwać dane oraz tabele
5. Transakcje i współbieżność	Co się dzieje, gdy dwie sesje pracują na tych samych danych naraz
6. Wydajność	Dlaczego zapytanie jest wolne, czym jest indeks i jak czytać plan wykonania
7. Eksploatacja	Jak zrobić kopię zapasową, komu dać dostęp i co monitorować
8. Słowniczek	Gdzie sprawdzić jedno zapomniane pojęcie albo skrót: Słowniczek pojęć bazodanowych, Skróty i akronimy, Mapa różnic między silnikami

Powiązane: Baza danych, silnik, instancja, schemat · Skąd się wziął SQL i dlaczego każdy silnik mówi nim inaczej · Który silnik bazy do czego: przewodnik decyzyjny

Źródła: Edgar F. Codd, „A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, czerwiec 1970 · en.wikipedia.org/wiki/Edgar_F._Codd · ibm.com/history/relational-database · content.dsp.co.uk, „The History of Oracle: 1977-2000" · en.wikipedia.org/wiki/History_of_Microsoft_SQL_Server · postgresql.org/docs/current/history.html · en.wikipedia.org/wiki/MySQL · en.wikipedia.org/wiki/NoSQL

Jak korzystać z tej książki

> Ta strona tłumaczy układ całej półki „Bazy danych" i to, czego można się spodziewać po każdej stronie — czytamy ją raz, na początku, i wracamy do niej, kiedy któraś konwencja się zapomni.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
ASCII	American Standard Code for Information Interchange: podstawowy zestaw znaków (litery, cyfry) używany w komputerach
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
HANA	SAP HANA: baza danych w pamięci firmy SAP, używana do dużych systemów księgowych i magazynowych
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
SAP	Systeme, Anwendungen und Produkte in der Datenverarbeitung: niemiecki producent oprogramowania klasy ERP, właściciel silnika HANA i przejętego wcześniej Sybase ASE
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
URL	Uniform Resource Locator: adres strony lub zasobu

Dlaczego jest osobna książka „fundamenty"

Silników baz danych na tej półce jest osiem — PostgreSQL, Oracle, MSSQL, MySQL, Sybase, MongoDB, Cassandra, SAP HANA. Większość tego, co administrator musi rozumieć, jest między nimi wspólna: czym jest klucz obcy, na czym polega złączenie, co oznacza poziom izolacji `READ COMMITTED`. Gdyby każda książka silnika tłumaczyła te pojęcia od nowa, powstałoby osiem wersji tej samej definicji, każda odrobinę inna — i za rok nikt by nie wiedział, która jest aktualna.

Dlatego teoria wspólna dla wszystkich silników trafia w jedno miejsce — do tej książki. Książki per silnik trzymają tylko to, co jest dla danego silnika specyficzne: konkretną składnię (`LIMIT` w PostgreSQL, `TOP` w MSSQL, `ROWNUM` albo `FETCH FIRST` w Oracle), narzędzia CLI (`psql`, `sqlplus`, `sqlcmd`, `mysql`), pliki konfiguracyjne, procedury instalacji i eksploatacji.

Strona silnikowa, która korzysta z pojęcia wyjaśnionego tutaj, nie tłumaczy go drugi raz — linkuje. Strona `INNER JOIN w PostgreSQL` zakłada, że wiemy, czym jest złączenie i skąd bierze się iloczyn kartezjański, bo to jest wytłumaczone w fundamentach, i skupia się wyłącznie na tym, jak to się pisze w PostgreSQL.

Ta książka jest silnikowo neutralna. Tam, gdzie silniki się różnią — a w warstwie SQL różnią się bardzo często — strona pokazuje tabelę porównawczą zamiast zdania „w zależności od silnika bywa różnie".

Sekcje strony

Każda strona objaśniająca (`typ/objasnienie`) ma ten sam zestaw sekcji, w tej samej kolejności. Nie każda ma wszystkie — na stronach czysto teoretycznych, jak te w tym rozdziale, sekcja `Fiszka` bywa krótka albo jej nie ma wcale, bo nie zawsze jest komenda do pokazania.

Fiszka. Komendy do skopiowania, jedna na nagłówek. Każda ma własny blok kodu i własny akapit opisu, a jeśli komenda coś wypisuje — własny blok z przykładowym wynikiem, oznaczony słowem `Przykład:`. To jest część do szybkiego zerknięcia, kiedy już wiemy, czego szukamy, i chcemy tylko sprawdzić dokładną składnię.

Po co to istnieje. Problem, który dane pojęcie albo mechanizm rozwiązuje. Tu pada też wyjaśnienie nowych terminów — akapit zaczynający się od `Pojęcie:` w ramce, w miejscu, w którym termin pojawia się pierwszy raz na stronie.

Analogia. Porównanie do czegoś spoza świata baz danych, rozwinięte na kilka zdań, razem z miejscem, w którym ta analogia się łamie — bo każda analogia się w końcu łamie, i warto wiedzieć gdzie.

Jak to działa. Mechanizm, na tyle głęboko, żeby zachowanie przestało zaskakiwać. To jest sekcja dla kogoś, kto już wie, co robi dana rzecz, i chce wiedzieć, dlaczego działa akurat tak.

Częste błędy. Konkretne pomyłki, z komunikatem błędu, który się wtedy widzi, i sposobem naprawy. Nie teoretyczne ostrzeżenia — sytuacje, które się naprawdę zdarzają.

Ćwiczenia. Odcięte poziomą kreską od reszty treści. Kto przyszedł po definicję albo komendę, może je pominąć bez czytania. Kto się uczy, znajduje tu zadania sprawdzające zrozumienie, zwykle z ukrytą odpowiedzią pod rozwijanym polem.

Tagi

Każda strona na tej półce ma tag typu, który mówi, czego się po niej spodziewać, zanim jeszcze zacznie się czytać treść.

`typ/objasnienie` — pojęcie albo mechanizm, który trzeba zrozumieć, nie tylko wykonać. Strony w tym rozdziale są tego typu. Mają pełny zestaw sekcji opisany wyżej.

`typ/procedura` — doprowadzenie systemu do konkretnego, jednego stanu: instalacja silnika, migracja bazy, wystawienie repliki. Bez analogii i bez ćwiczeń, bo kto to czyta, ma zadanie do wykonania, a nie ochotę na naukę przy okazji. Procedura zawsze linkuje do strony objaśniającej, żeby ten, kto chce wiedzieć więcej, miał dokąd pójść.

`typ/referencja` — tabela do przeszukiwania wzrokiem: mapa typów danych między silnikami, lista kodów błędów, zestawienie poziomów izolacji. Bez narracji, bo narracja tylko by tu przeszkadzała.

Do tego dochodzą tagi tematyczne (`temat/klucze`, `temat/relacje`, `temat/transakcje`...) i tag poziomu (`poziom/podstawy`, `poziom/srednio-zaawansowany`), które pozwalają przefiltrować książkę według tematu albo doświadczenia czytelnika, niezależnie od tego, w jakim rozdziale strona akurat leży.

Trzy warstwy tagów

Każda strona nosi tagi z trzech niezależnych od siebie kategorii naraz, i wszystkie trzy razem pozwalają znaleźć stronę, nie znając jej dokładnego tytułu ani rozdziału.

Warstwa	Przykłady	Co filtruje
Typ	`typ/objasnienie`, `typ/procedura`, `typ/referencja`	jakiego kształtu strony szukamy
Poziom	`poziom/podstawy`, `poziom/srednio-zaawansowany`	ile już wiemy o temacie
Temat	`temat/klucze`, `temat/relacje`, `temat/transakcje`, `temat/wydajnosc`	czego dotyczy treść

Strona `Klucze — główny, obcy, unikalny, naturalny, sztuczny` nosi więc jednocześnie `typ/objasnienie`, `poziom/podstawy` i `temat/klucze` — trzy niezależne informacje o tej samej stronie, każda przydatna w innej sytuacji wyszukiwania.

Placeholdery i przykłady

Strony objaśniające w tej książce, w przeciwieństwie do stron proceduralnych na pozostałych półkach, częściej pokazują konkretne przykładowe nazwy (`zamowienia`, `klienci`, `produkty`) niż polskie placeholdery w stylu `NazwaTabeli`. Wynika to z charakteru materiału: strona proceduralna prowadzi czytelnika przez komendy do wklejenia na jego własnym systemie, więc placeholder mówi wprost, co tam wstawić. Strona teoretyczna pokazuje działający, samodzielny przykład — CREATE TABLE z konkretnym, spójnym schematem, do którego wraca kilka stron pod rząd, żeby czytelnik

nie musiał za każdym razem wyobrażać sobie nowych tabel.

Blok z wynikiem komendy jest zawsze oznaczony słowem `Przykład:` i pokazuje realistyczne, wiarygodne dane — takie, jakie zobaczylibyśmy naprawdę w `psql`, `sqlplus`, `sqlcmd` albo kliencie `mysql`, z prawdziwym formatowaniem ramek i nagłówków kolumn właściwych dla danego silnika.

Anonimizacja i linkowanie

Nazwy serwerów, baz i kont w przykładach pochodzą z fikcyjnej firmy Zielona sp. z o.o., opisanej na stronie Świat Zielonej: firma, w której dzieją się przykłady: MSSQL na `sql01` z bazą `CDN_Zielona`, Oracle na `ora01` z bazą `WMSPRD`, PostgreSQL na `pg01`. W komendach do skopiowania stoją polskie placeholdery (`NazwaBazy`, `NazwaSerwera`), które podmieniamy na swoje.

Odwołanie do innej strony tej książki albo innej książki na tej samej półce zapisujemy jako `[Tytuł strony](#)` — link w nawiasie zostaje pusty, bo prawdziwy adres URL dogenerowuje proces budowania książki na podstawie tytułu. Dlatego tytuły stron w tej książce nie zmieniają się bez powodu: zmiana tytułu psuje każdy link, który się do niego odwoływał.

Diagramy i tabele porównawcze

Tam, gdzie silniki różnią się od siebie w sposób, który trudno opisać jednym zdaniem, strona pokazuje tabelę — kolumna na silnik, wiersz na cechę — zamiast zdania „w zależności od silnika bywa różnie". Tabela pozwala porównać cztery silniki jednym rzutem oka; zdanie tego nie robi.

Tam, gdzie relacja między obiektami (tabelami, procesami, warstwami architektury) jest łatwiejsza do zrozumienia na obrazku niż w prozie, strona pokazuje prosty diagram tekstowy w bloku ```` ```text ```` , złożony ze strzałek i ramek rysowanych znakami ASCII. To nie jest ozdobnik — to sposób pokazania kierunku zależności (kto kogo referuje, co jest stroną „jeden", a co stroną „wiele"), który w samym tekście łatwo przeoczyć.

Numeracja plików i stron

Pliki wewnątrz jednego rozdziału są numerowane w kolejności czytania (`01-`, `02-`, `03-`...) — ta numeracja odpowiada numeracji z planu półki i pilnuje kolejności, w jakiej strony trafiają do książki podczas budowania. Numer w nazwie pliku nie jest widoczny dla czytelnika BookStacka — widzi on wyłącznie tytuł strony (nagłówek `#`) i jej miejsce w spisie treści książki, ustalone przez rozdział, do którego strona należy.

Jak z tego korzystać

Rozdziały 1–8 tej książki czyta się w miarę liniowo — każdy zakłada znajomość poprzedniego, bo pojęcie zdefiniowane raz nie jest definiowane drugi raz. Rozdział 8, słowniczek, jest wyjątkiem: to strona do wyszukiwania, nie do czytania od deski do deski.

Strony silnikowe odwołują się do fundamentów linkiem w tekście albo w sekcji `Powiązane:` na dole strony. Warto kliknąć, kiedy pojawia się pojęcie, którego nie pamiętamy dokładnie — strona silnikowa nie powtórzy definicji, bo ta już tu jest.

Baza danych, silnik, instancja, schemat: cztery pojęcia, które się myli

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

Cztery słowa, których używamy zamiennie w rozmowie, a które w każdym silniku znaczą coś innego — i to jest dokładnie ta różnica, która myli ludzi migrujących między silnikami.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
RAM	Random Access Memory: pamięć operacyjna
SGA	System Global Area: wspólna pamięć instancji Oracle
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

TYLKO ODCZYT

Sprawdzenie bieżącej bazy w PostgreSQL

```
SELECT current_database();
```

Zwraca nazwę bazy, do której jest aktualnie otwarte połączenie.

Przykład:

```
 current_database
------------------
 produkcja
(1 row)
```

□□ TYLKO ODCZYT

Sprawdzenie bieżącej bazy w MSSQL

```
SELECT DB_NAME();
```

Przykład (sqlcmd):

```
-------------------------------------------------------------
ZamowieniaProd


(1 rows affected)
```

□□ **MSSQL** = Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima

□□ TYLKO ODCZYT

Sprawdzenie bieżącej bazy w Oracle

```
SELECT name FROM v$database;
```

W Oracle nie ma odpowiednika `USE nazwa_bazy` — połączenie jest zestawiane od razu z konkretną instancją, a ta ma dokładnie jedną bazę.

Przykład (sqlplus):

```
NAME
---------
ORCLDB


1 row selected.
```

TYLKO ODCZYT

Sprawdzenie bieżącej bazy w MySQL

```
SELECT DATABASE();
```

Przykład (klient mysql):

```
+------------+
| DATABASE() |
+------------+
| sklep      |
+------------+
1 row in set (0.00 sec)
```

TYLKO ODCZYT

Sprawdzenie bieżącego schematu w PostgreSQL

```
SELECT current_schema();
```

Przykład:

```
 current_schema
----------------
 public
(1 row)
```

TYLKO ODCZYT

Sprawdzenie bieżącego schematu w MSSQL

```
SELECT SCHEMA_NAME();
```

Przykład:

```
------------------------------------------------------------
dbo
```

```
(1 rows affected)
```

MSSQL = Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima

TYLKO ODCZYT

Sprawdzenie bieżącego schematu w Oracle

```
SELECT sys_context('USERENV', 'CURRENT_SCHEMA') FROM dual;
```

W Oracle schemat jest tożsamy z użytkownikiem, więc ta wartość zwykle pokrywa się z loginem, którym się połączyliśmy.

Przykład:

```
SYS_CONTEXT('USERENV','CURRENT_SCHEMA')
----------------------------------------
SKLEP


1 row selected.
```

W MySQL osobnego zapytania o schemat nie ma — `SELECT DATABASE()` z Fiszki wyżej jest jednocześnie odpowiedzią na pytanie o schemat, bo to w tym silniku dokładnie to samo pojęcie.

Po co to istnieje

Cztery słowa — baza danych, silnik, instancja, schemat — słyszy się w rozmowie o bazach non stop, często jako synonimy. „Zrestartuj bazę", „wejdź do schematu", „ta instancja". Dopóki pracujemy z jednym silnikiem codziennie, nieprecyzyjne słownictwo nie szkodzi, bo w głowie mamy jeden konkretny model. Problem pojawia się w chwili, gdy trzeba przełączyć się na inny silnik albo przeczytać dokumentację drugiego — bo te cztery słowa nie mapują się między silnikami jeden do jednego, i przeniesienie nawyku myślowego z jednego na drugi prowadzi do błędnych założeń.



Silnik uruchamia instancję, instancja otwiera bazę danych, a baza dzieli się na schematy (kliknięcie otwiera pełny rozmiar)

Pojęcie: baza danych. Zorganizowany zbiór danych trzymany trwale na dysku — same dane, ich struktura i ograniczenia, bez procesów, które je obsługują. Fizycznie jest to zestaw plików: danych, indeksów, logów transakcyjnych.

Pojęcie: silnik bazodanowy (DBMS). Oprogramowanie, które umie czytać i zapisywać taki zbiór danych zgodnie z regułami — SQL-em, transakcjami, uprawnieniami. PostgreSQL, Oracle Database, Microsoft SQL Server i MySQL to cztery różne silniki. Silnik to produkt, który instalujemy raz; z tego samego zainstalowanego silnika można uruchomić wiele działających kopii.

Pojęcie: instancja. Uruchomiona, działająca w pamięci kopia silnika — procesy systemowe plus struktury w pamięci RAM (bufory, cache planów zapytań), które razem obsługują żądania klientów. Instancję się startuje i zatrzymuje; bazę danych się otwiera i zamyka. To rozróżnienie w PostgreSQL i MySQL jest subtelne, bo tam start instancji i otwarcie bazy to praktycznie ta sama czynność. W Oracle jest to rozróżnienie fundamentalne, patrz tabela niżej.

Pojęcie: schemat. Przestrzeń nazw wewnątrz bazy danych, która grupuje tabele, widoki, procedury i inne obiekty, żeby dwie tabele o tej samej nazwie mogły istnieć obok siebie bez konfliktu (`sprzedaz.klienci` i `archiwum.klienci`). Schemat nie jest osobnym zbiorem plików — to etykieta na obiektach wewnątrz jednej bazy.

Jak to mapuje się między silnikami

Poniższa tabela jest sercem tej strony — te cztery pojęcia realizują się w czterech głównych silnikach na cztery różne sposoby.

Warstwa	PostgreSQL	Oracle	MSSQL	MySQL
Silnik	proces `postgres` zainstalowany raz na serwerze	Oracle Database, zainstalowany raz (`ORACLE_HOME`)	usługa Windows/Linux `MSSQLSERVER` lub nazwana instancja	proces `mysqld` zainstalowany raz na serwerze

Warstwa	PostgreSQL	Oracle	MSSQL	MySQL
Instancja	jeden `postmaster` obsługuje wiele baz jednocześnie	pamięć współdzielona (SGA) + procesy tła (`PMON`, `SMON`, `DBWn`, `LGWR`) — w klasycznym modelu jedna instancja otwiera dokładnie jedną bazę	jedna uruchomiona usługa SQL Server; na jednym serwerze można zainstalować kilka nazwanych instancji obok siebie	jeden proces `mysqld` obsługuje wiele baz jednocześnie
Baza danych	jeden katalog danych wewnątrz klastra, ma własne schematy	zestaw plików fizycznych (datafiles, control files, redo logs) montowanych i otwieranych przez instancję	jeden zestaw plików `.mdf` / `.ldf` wewnątrz instancji, ma własne schematy	w MySQL baza danych i schemat to jedno i to samo pojęcie
Schemat	przestrzeń nazw wewnątrz bazy (`public` domyślnie)	w Oracle schemat jest tożsamy z użytkownikiem — `CREATE USER` tworzy jednocześnie schemat o tej samej nazwie	przestrzeń nazw wewnątrz bazy (`dbo` domyślnie), niezależna od loginu	brak osobnego pojęcia — `CREATE SCHEMA` jest w MySQL synonimem `CREATE DATABASE`

Dwa wiersze tej tabeli tłumaczą większość nieporozumień. Pierwszy: w Oracle „instancja" i „baza" to dwa byty połączone relacją bliską 1:1 w klasycznej architekturze — instancja to pamięć i procesy, baza to pliki na dysku, a restart instancji nie kasuje bazy, tylko zamyka i ponownie otwiera dostęp do tych samych plików. Drugi: w MySQL słowa „baza" i „schemat" są dosłownymi synonimami, więc pytanie „ile schematów ma ta baza" nie ma w MySQL sensu — baza jest schematem.

Analogia

Wyobraźmy sobie drukarnię. Silnik to model konkretnej maszyny drukarskiej — projekt, który da się kupić i postawić w dowolnej hali. Instancja to ta sama maszyna, ale już fizycznie postawiona w konkretnej hali, podłączona do prądu i włączona — z rozgrzanymi rolkami i załadowanym papierem, gotowa przyjmować zlecenia. Baza danych to stos gotowych dokumentów, które ta maszyna produkuje i przechowuje w swoim magazynie. Schemat to półki w tym magazynie, na które dokumenty są rozłożone według działu.

Tu analogia zaczyna oddawać różnicę między silnikami. PostgreSQL i MySQL to hala z jedną maszyną, która obsługuje jednocześnie kilka niezależnych magazynów (baz) — operator podchodzi do jednej maszyny i mówi, z którym magazynem chce pracować. Oracle w klasycznej architekturze to raczej jedna maszyna przypisana na stałe do jednego magazynu — jeśli potrzebujemy drugiego magazynu, stawiamy drugą maszynę (drugą instancję), a nie przełączamy istniejącą. MSSQL jest pośrodku: jedna usługa (instancja) obsługuje wiele magazynów (baz), ale na tym samym serwerze fizycznym można też postawić kilka odrębnych usług (nazwanych instancji) obok siebie, każdą z

własnym zestawem magazynów.

Analogia łamie się przy schemacie w Oracle — tam „półka w magazynie" jest jednocześnie „osobą, która ma do tej półki klucz". Schemat i użytkownik to w Oracle jeden byt, co w pozostałych trzech silnikach nie jest prawdą: tam użytkownik i schemat, do którego ten użytkownik ma dostęp, to dwie osobne rzeczy.

Jak to działa

W PostgreSQL uruchomienie `pg_ctl start` startuje jeden proces nadzorczy (`postmaster`), który następnie sam odpala procesy potomne obsługujące poszczególne połączenia. Ten sam postmaster widzi wszystkie bazy w klastrze danych naraz, ale jedno połączenie klienckie jest zawsze przywiązane do dokładnie jednej bazy na cały czas swojego trwania — nie da się w trakcie sesji przełączyć na inną bazę komendą SQL, trzeba się rozłączyć i połączyć ponownie ze wskazaniem innej bazy w parametrach połączenia. Wewnątrz tej jednej bazy `SET search_path` przełącza aktywny schemat swobodnie, bo schematy nie są osobnymi połączeniami.

W MySQL rozróżnienia między bazą a schematem nie ma na poziomie silnika w ogóle — `CREATE DATABASE nazwa` i `CREATE SCHEMA nazwa` wykonują dokładnie ten sam kod. Przełączenie kontekstu w trakcie sesji jest za to trywialne: `USE nazwa_bazy` działa bez rozłączania.

W MSSQL jedna instancja może hostować wiele baz, a przełączenie kontekstu też jest jedną komendą (`USE NazwaBazy`), bo to wciąż to samo połączenie do tej samej usługi. Odwołanie się do obiektu w innej bazie bez przełączania kontekstu jest możliwe przez pełną nazwę trójczłonową: `NazwaBazy.dbo.NazwaTabeli` — czego PostgreSQL między bazami wprost nie robi.

W Oracle instancja startuje niezależnie od tego, czy baza jest otwarta — `STARTUP NOMOUNT` uruchamia SGA i procesy tła bez montowania jakiejkolwiek bazy, `STARTUP MOUNT` czyta plik kontrolny i montuje bazę, `STARTUP` (bez parametru) od razu ją otwiera. Ten podział na trzy etapy nie ma odpowiednika w pozostałych trzech silnikach i jest bezpośrednią konsekwencją rozdzielenia instancji od bazy na poziomie architektury.

Uwaga: Oracle wielobazowy (CDB/PDB)

Od wersji 12c Oracle wprowadził architekturę wielobazową (multitenant), w której jedna instancja może obsługiwać jedną bazę kontenerową (`CDB`) mieszczącą w sobie wiele niezależnych baz podłączanych (`PDB`, pluggable database) — każda ze swoim własnym zestawem schematów, odizolowana od pozostałych. To zbliża Oracle do modelu PostgreSQL czy MySQL, gdzie jedna

instancja obsługuje wiele baz naraz, ale relacja instancja-baza w klasycznym, niewielobazowym Oracle — wciąż powszechnym na starszych środowiskach — pozostaje 1:1, i to jej dotyczy reszta tej strony.

Częste błędy

Traktowanie „instancji" w Oracle jak restartu usługi w MSSQL. Restart instancji Oracle (`SHUTDOWN` + `STARTUP`) zamyka i ponownie otwiera dostęp do plików bazy — to jest bliższe zatrzymaniu i uruchomieniu usługi `MSSQLSERVER` niż potocznemu „zrestartuj bazę" w PostgreSQL, gdzie restart usługi też dotyczy wszystkich baz naraz, bo jest tylko jeden proces nadzorczy na cały klaster.

Szukanie `USE NazwaBazy` w PostgreSQL. Taka komenda nie istnieje — w PostgreSQL zmiana bazy wymaga nowego połączenia, nie polecenia SQL w istniejącej sesji. Próba wykonania `\c nazwa_bazy` w skrypcie SQL (a nie w `psql`) kończy się błędem składni, bo `\c` jest poleceniem klienta `psql`, nie SQL-em.

Filtrowanie zapytań do `information_schema` bez `table_schema` w bazie, gdzie „schemat" i „baza" to nie to samo. Zapytanie napisane z myślą o MySQL, gdzie schemat i baza się pokrywają, po przeniesieniu do PostgreSQL czy Oracle zwraca wiersze z niewłaściwego schematu, bo tam jedna baza rutynowo trzyma kilka schematów naraz.

□□ Ćwiczenia

> Poniższe zadania wykonujemy na dowolnym zainstalowanym silniku, do którego mamy dostęp — nie trzeba mieć wszystkich czterech.

Ćwiczenie 1 — policz warstwy

Na silniku, który mamy pod ręką, sprawdzamy po kolei: nazwę silnika i jego wersję, nazwę instancji (jeśli silnik to pojęcie udostępnia), nazwę bieżącej bazy, nazwę bieżącego schematu.

Odpowiedź

W PostgreSQL: `SELECT version();`, `SELECT current_database();`, `SHOW search_path;`. Osobnej „nazwy instancji" PostgreSQL nie eksponuje jako pojedynczej wartości — tożsamość instancji to po prostu katalog danych (`SHOW data_directory;`).

W MSSQL: `SELECT @@SERVERNAME;` dla instancji, `SELECT DB_NAME();` dla bazy, `SELECT SCHEMA_NAME();` dla schematu.

Pytania kontrolne

1. Dlaczego w Oracle da się uruchomić instancję bez otwierania żadnej bazy, a w MySQL nie da się uruchomić `mysqld` bez żadnej bazy dostępnej do pracy?

Odpowiedź

Bo w Oracle instancja (pamięć i procesy) i baza (pliki na dysku) to dwa osobne byty połączone przez montowanie — instancja może istnieć „na sucho", zanim wskaże jej się bazę. W MySQL nie ma takiego rozdzielenia: proces `mysqld` od startu operuje na katalogu danych, w którym bazy systemowe (`mysql`, `information_schema`) muszą istnieć, żeby serwer w ogóle wystartował.

2. W Oracle utworzenie nowego użytkownika (`CREATE USER`) automatycznie tworzy też nowy schemat. Dlaczego w PostgreSQL trzeba to zrobić dwiema osobnymi komendami?

Odpowiedź

Bo w Oracle użytkownik i schemat to ten sam byt pod dwiema nazwami — nie da się mieć jednego bez drugiego. W PostgreSQL użytkownik (rola z prawem logowania) i schemat (przestrzeń nazw obiektów) są od siebie niezależne: jeden użytkownik może mieć dostęp do wielu schematów, a schemat może istnieć bez żadnego dedykowanego dla niego użytkownika.

Powiązane: Jak korzystać z tej książki · Model relacyjny: tabela, wiersz, kolumna, dziedzina · Katalog systemowy: skąd baza wie, co w sobie ma

Źródła: dokumentacja PostgreSQL (rozdz. „Database Roles and Privileges"), Oracle Database Concepts (rozdz. „Database and Instance Configuration"), Microsoft Docs „Database Engine Instances", MySQL Reference Manual (rozdz. „The MySQL Data Dictionary")

Model relacyjny: tabela, wiersz, kolumna, dziedzina

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

Tabela, wiersz i kolumna wydają się oczywiste, dopóki nie trzeba wytłumaczyć, dlaczego zapytanie bez `ORDER BY` może za każdym razem zwrócić dane w innej kolejności.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
IBM	International Business Machines: firma, w której laboratorium powstał model relacyjny i pierwszy działający silnik SQL
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

TYLKO ODCZYT

Zapytanie bez gwarancji kolejności

```
SELECT id, nazwa FROM produkty;
```

Silnik nie ma obowiązku zwrócić wierszy w kolejności, w jakiej je wstawiliśmy — zwraca je w kolejności, która akurat jest dla niego najwygodniejsza.

Przykład:

```
 id |    nazwa
----+------------
  3 | Klawiatura
```

```
  1 | Monitor
  2 | Mysz
(3 rows)
```

TYLKO ODCZYT

Zapytanie z jawną kolejnością

```
SELECT id, nazwa FROM produkty ORDER BY id;
```

Jedyny sposób, żeby kolejność wyniku była gwarantowana i powtarzalna.

Przykład:

```
 id |    nazwa
----+------------
  1 | Monitor
  2 | Mysz
  3 | Klawiatura
(3 rows)
```

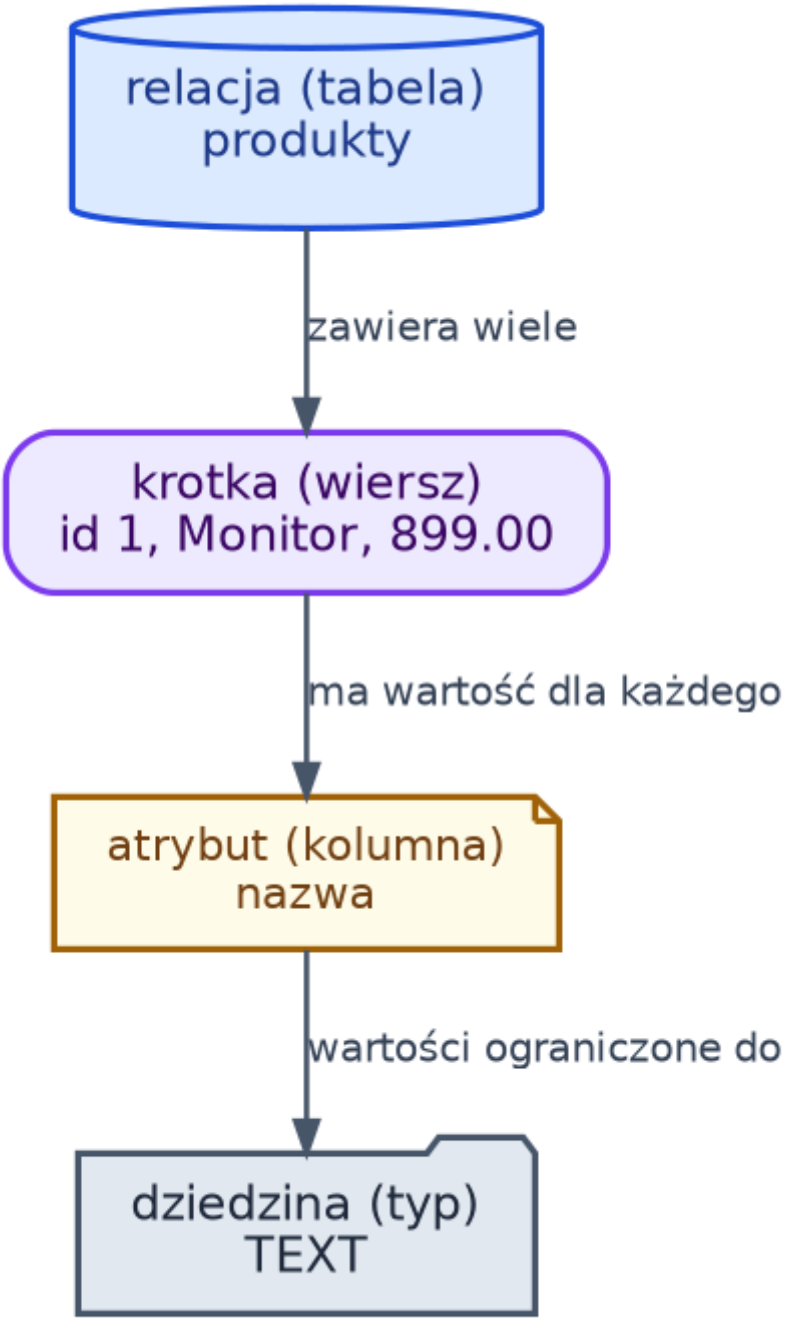
Po co to istnieje

Model relacyjny jest matematyczną podstawą, na której stoi każdy z silników SQL na tej półce — PostgreSQL, Oracle, MSSQL, MySQL, Sybase. Zanim zaczniemy pisać `SELECT` czy `JOIN`, warto wiedzieć, skąd biorą się słowa „tabela", „wiersz" i „kolumna" i dlaczego niektóre ich własności — jak brak gwarantowanej kolejności — nie są usterką konkretnego silnika, tylko wynikają wprost z definicji sprzed ponad pięćdziesięciu lat.

Model relacyjny opisał w 1970 roku Edgar F. Codd, matematyk pracujący w IBM, w artykule „A Relational Model of Data for Large Shared Data Banks". Ówczesne bazy danych (hierarchiczne i sieciowe) wymagały od programisty znajomości fizycznej struktury danych — żeby coś odczytać, trzeba było wiedzieć, jak dane są powiązane wskaźnikami na dysku. Codd zaproponował coś innego: dane opisane wyłącznie w kategoriach matematycznej teorii zbiorów i logiki predykatów, gdzie sposób zapytania o dane jest całkowicie oddzielony od sposobu ich fizycznego przechowywania. Pierwsze komercyjne wdrożenia tego pomysłu — w tym prototyp SQL — powstały w IBM i Oracle (wtedy jeszcze pod nazwą Relational Software) w drugiej połowie lat siedemdziesiątych.

Relacja, krotka, dziedzina — pojęcia matematyczne pod spodem

Pojęcie: relacja. W sensie matematycznym, w jakim używał tego słowa Codd, relacja to zbiór krotek o jednakowej strukturze — dokładnie to, co w praktyce nazywamy tabelą. To słowo wraca też w innym znaczeniu — jako „relacja między tabelami" (klucz obcy, powiązanie 1:N) — i to jest inne pojęcie, opisane osobno na stronie Relacje: 1:1, 1:N, N:M i tabela łącząca. Tu, mówiąc „relacja", mamy na myśli tabelę.



Relacja zawiera krotki, każda krotka ma wartość dla każdego atrybutu, a wartości atrybutu ogranicza jego dziedzina (kliknięcie otwiera pełny rozmiar)

> **Pojęcie: krotka (ang. tuple).** Pojedynczy element relacji — w praktyce jeden wiersz tabeli. Matematycznie krotka to uporządkowany zestaw wartości, po jednej z każdej dziedziny relacji.

Pojęcie: dziedzina (ang. domain). Zbiór dopuszczalnych wartości dla danej kolumny — w praktyce to, co nazywamy typem danych kolumny (`INTEGER`, `DATE`, `VARCHAR(50)`). Dziedzina ogranicza, co w ogóle może się znaleźć w danej pozycji krotki — kolumna zadeklarowana jako `DATE` nie przyjmie tekstu „poniedziałek".

Pojęcie: kardynalność relacji. Liczba krotek w relacji — czyli, w praktyce, liczba wierszy tabeli. Nie mylić z kardynalnością relacji między tabelami (1:1, 1:N, N:M) — to inne użycie tego samego słowa, opisane na stronie o relacjach.

Pojęcie: stopień relacji (ang. degree, arity). Liczba atrybutów (kolumn) relacji. Tabela o pięciu kolumnach ma stopień piąty, niezależnie od tego, ile ma wierszy.

Pojęcie: schemat relacji. Nazwa relacji razem z listą jej atrybutów i ich dziedzin — czyli, w praktyce, to, co zapisujemy w `CREATE TABLE`: definicja, jak tabela ma wyglądać, zanim jeszcze trafi do niej choć jeden wiersz. Odróżnia się od instancji relacji, czyli faktycznego zestawu krotek znajdujących się w tabeli w danym momencie. To rozróżnienie tłumaczy, dlaczego `ALTER TABLE` (zmiana schematu relacji) i `INSERT`/`UPDATE`/`DELETE` (zmiana instancji relacji) są operacjami tak różnej wagi — pierwsza zmienia definicję dla wszystkich przyszłych i istniejących wierszy naraz, druga dotyczy tylko konkretnych danych.

Pojęcie: atrybut. Nazwana kolumna relacji — w czystym modelu matematycznym atrybuty, podobnie jak krotki w relacji, nie mają ustalonej kolejności; odwołujemy się do nich po nazwie, nie po pozycji. SQL tej czystości nie zachowuje: kolumny tabeli mają zdefiniowaną kolejność zapisaną przy `CREATE TABLE`, `SELECT *` zwraca je zawsze w tej samej kolejności, a `INSERT` bez jawnej listy kolumn polega właśnie na tej kolejności.

Czysta relacja w sensie matematycznym ma dwie własności, których tabela SQL formalnie nie musi mieć. Po pierwsze, relacja jest zbiorem, więc nie zawiera duplikatów — dwie identyczne krotki to wciąż jeden element zbioru. Tabela SQL bez klucza głównego duplikaty jak najbardziej dopuszcza; `SELECT * FROM tabela` może zwrócić dwa identyczne wiersze. Po drugie, zbiór z definicji nie ma kolejności elementów — a to prowadzi wprost do sekcji niżej.

Analogia

Tabela relacyjna przypomina arkusz kalkulacyjny — wiersze, kolumny, komórka na przecięciu. To porównanie działa dobrze jako pierwsze wprowadzenie i tłumaczy, dlaczego w ogóle mówimy o „wierszach" i „kolumnach", a nie o czymś bardziej abstrakcyjnym.

Ale porównanie zaczyna zawodzić dość szybko. W arkuszu każda komórka może mieć inny format i inną zawartość niezależnie od reszty kolumny — jedna komórka liczbę, sąsiednia w tej samej kolumnie tekst albo formułę. W tabeli relacyjnej cała kolumna dzieli jedną dziedzinę: jeśli kolumna jest typu `INTEGER`, każda wartość w niej jest liczbą całkowitą, bez wyjątków. Arkusz nie ma też pojęcia klucza — nic nie stoi na przeszkodzie, żeby dwa wiersze były identyczne, i arkusz się tym nie przejmuje; tabela z kluczem głównym fizycznie nie pozwoli takiej sytuacji powstać.

Największa różnica dotyczy kolejności. W arkuszu kolejność wierszy jest ich integralną częścią — przesuwamy wiersz siódmy nad trzeci przeciągnięciem myszką, i to jest trwała zmiana zapisana w pliku. W tabeli relacyjnej kolejność wierszy nie jest w ogóle częścią danych. To jest najczęstsze zaskoczenie osób, które uczą się SQL po latach pracy w arkuszach.

Jak to działa

Relacja w sensie matematycznym jest zbiorem, a zbiór z definicji nie ma kolejności — {1, 2, 3} i {3, 1, 2} to ten sam zbiór. SQL dziedziczy tę własność: `SELECT` bez klauzuli `ORDER BY` nie daje żadnej gwarancji co do kolejności zwracanych wierszy, niezależnie od silnika.

W praktyce silnik musi fizycznie przechowywać wiersze w jakiejś kolejności na dysku, więc zapytanie bez `ORDER BY` zwykle zwraca dane w jakiejś powtarzalnej kolejności — najczęściej zbliżonej do kolejności zapisu na stronach danych. To złudzenie stabilności jest niebezpieczne, bo ta kolejność potrafi się zmienić bez żadnej zmiany w danych: po przebudowie indeksu, po `VACUUM FULL` w PostgreSQL, po tym jak optymalizator wybierze inny plan wykonania (skan indeksu zamiast skanu sekwencyjnego), albo po prostu po aktualizacji wersji silnika. Kod, który zakłada konkretną kolejność bez `ORDER BY`, jest kodem, który działa do pierwszej migracji albo pierwszego większego `UPDATE`.

To samo dotyczy stronicowania wyników. Klauzula `LIMIT 10 OFFSET 20` (albo jej odpowiednik w innym silniku) bez `ORDER BY` może zwrócić na stronie drugiej część wierszy, które już były na stronie pierwszej, oraz pominąć inne — bo silnik nie obiecał żadnej konkretnej kolejności, na której stronicowanie mogłoby się oprzeć.

Częste błędy

Poleganie na kolejności kolumn zamiast na ich nazwach. `INSERT INTO produkty VALUES (1, 'Monitor', 899.00)` działa tylko tak długo, jak nikt nie doda nowej kolumny ani nie zmieni kolejności istniejących. Dodanie kolumny `kategoria_id` w środku definicji tabeli przesuwa pozycje wszystkich kolumn po niej, a taki `INSERT` zacznie wstawiać cenę tam, gdzie wcześniej była nazwa. Bezpieczna forma zawsze wymienia kolumny wprost: `INSERT INTO produkty (id, nazwa, cena) VALUES (...)`.

Poleganie na kolejności wstawiania bez `ORDER BY`. Zapytanie testowane na małej tabeli deweloperskiej zwraca wiersze „w kolejności, w jakiej je dodaliśmy", więc programista zakłada, że

tak będzie zawsze. Na produkcji, po naroście danych i przebudowie indeksów, ta sama kwerenda zaczyna zwracać wiersze w innej kolejności — i raport, który miał pokazywać „ostatnie dziesięć zamówień", zaczyna pokazywać przypadkowe dziesięć.

Zakładanie, że tabela SQL nigdy nie ma duplikatów. Bez klucza głównego albo ograniczenia `UNIQUE` nic nie broni wstawienia dwóch identycznych wierszy — `INSERT` wykonany dwa razy z tymi samymi wartościami po prostu doda drugi, identyczny wiersz. To zachowanie zgodne z tym, jak działa tabela SQL, ale niezgodne z czystą definicją relacji jako zbioru, i bywa źródłem błędów w raportach, które liczą wiersze zamiast unikalnych wartości.

Porównywanie `NULL` operatorem `=`. `NULL` reprezentuje brak wartości w dziedzinie, a nie jedną z jej wartości, więc `kolumna = NULL` nie zwraca ani prawdy, ani fałszu — zwraca `NULL`, co w klauzuli `WHERE` liczy się jako „nie pokazuj tego wiersza". Właściwym operatorem jest `IS NULL`. Temat wraca osobno przy pełnym omówieniu trójwartościowej logiki SQL.

□□ Ćwiczenia

> Poniższe zadania wykonujemy na dowolnym silniku SQL, do którego mamy dostęp.

Ćwiczenie 1 — brak gwarancji kolejności

```
CREATE TABLE test_kolejnosci (id INTEGER, nazwa TEXT);
INSERT INTO test_kolejnosci VALUES (3, 'c'), (1, 'a'), (2, 'b');
SELECT * FROM test_kolejnosci;
```

Uruchamiamy to zapytanie kilka razy z rzędu, a potem po dodaniu i usunięciu kilkuset dodatkowych wierszy. Czy kolejność wyniku się zmienia?

Odpowiedź

Na małej, świeżo wstawionej tabeli kolejność zwykle pozostaje zbliżona do kolejności wstawiania, bo silnik czyta dane po prostu w kolejności zapisu na dysku. Po większych operacjach `DELETE`/`INSERT`, `VACUUM`, przebudowie indeksu albo po prostu wzroście liczby wierszy powyżej progu, przy którym optymalizator zaczyna wybierać skan indeksu zamiast sekwencyjnego, kolejność może się zmienić — bo nic w definicji SQL jej nie gwarantowało.

Ćwiczenie 2 — atrybuty a pozycja kolumn

```
CREATE TABLE produkty (id INTEGER, nazwa TEXT, cena NUMERIC);
INSERT INTO produkty VALUES (1, 'Monitor', 899.00);
ALTER TABLE produkty ADD COLUMN kategoria_id INTEGER;
INSERT INTO produkty VALUES (2, 'Mysz', 79.00);
```

Co się stanie z drugim `INSERT` i dlaczego to jest niebezpieczne w kodzie produkcyjnym?

Odpowiedź

`ALTER TABLE ... ADD COLUMN` w większości silników dodaje nową kolumnę na końcu listy, więc pozycje istniejących kolumn się nie zmieniają i ten konkretny `INSERT` zadziała poprawnie — ale tylko dlatego, że nowa kolumna wylądowała na końcu, a nie w środku. Gdyby ktoś wstawił kolumnę między `nazwa` i `cena` (co niektóre silniki pozwalają zrobić przez przebudowanie tabeli), każdy `INSERT` bez jawnej listy kolumn zacząłby wstawiać wartości w złych miejscach bez żadnego ostrzeżenia ze strony silnika.

Pytania kontrolne

1. Czym różni się dziedzina kolumny od typu danych kolumny w SQL?

Odpowiedź

W praktyce, na poziomie pojedynczej kolumny w konkretnym silniku, to niemal to samo — typ danych jest realizacją dziedziny. Różnica pojawia się, gdy dziedzina jest węższa niż typ: kolumna `wiek INTEGER CHECK (wiek >= 0)` ma typ `INTEGER`, ale jej rzeczywista dziedzina to nieujemne liczby całkowite — ograniczenie `CHECK` domyka dziedzinę ciaśniej niż sam typ danych.

2. Dlaczego dwa identyczne pod względem wartości wiersze w tabeli SQL nie łamią żadnej reguły silnika, mimo że łamią definicję relacji jako zbioru?

Odpowiedź

Bo SQL jako język nie wymusza czystości modelu matematycznego — tabela bez klucza głównego ani ograniczenia `UNIQUE` jest w SQL wielozbiorem (ang. multiset albo bag), nie zbiorem. Uzyskanie prawdziwie relacyjnej semantyki „brak duplikatów" wymaga świadomego dodania ograniczenia, które to wymusi.

Powiązane: Baza danych, silnik, instancja, schemat: cztery pojęcia, które się myli · Klucze: główny, obcy, unikalny, naturalny, sztuczny · ORDER BY, LIMIT/TOP/ROWNUM/FETCH — porcjowanie wyniku

Źródła: E. F. Codd, „A Relational Model of Data for Large Shared Data Banks" (1970) · C. J. Date, „An Introduction to Database Systems" · dokumentacja PostgreSQL, rozdz. „Query Language (SQL)"

Klucze: główny, obcy, unikalny, naturalny, sztuczny

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

Klucz główny, klucz obcy i klucz unikalny brzmią jak trzy warianty tej samej rzeczy, a rozwiązują trzy różne problemy.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
PESEL	Powszechny Elektroniczny System Ewidencji Ludności: jedenastocyfrowy numer identyfikacyjny osoby w Polsce, przykład danych, które filtr danych osobowych powinien wykrywać
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
UUID	Universally Unique Identifier: długi, praktycznie niepowtarzalny identyfikator

Fiszka

ZMIENIA SYSTEM

Utworzenie tabeli z kluczem głównym

```
CREATE TABLE klienci (
    id SERIAL PRIMARY KEY,
    nazwa TEXT NOT NULL,
    nip TEXT UNIQUE
);
```

Kolumna `id` staje się kluczem głównym tabeli, a kolumna `nip` dostaje osobne ograniczenie unikalności.

Przykład:

```
CREATE TABLE
```

□□ **id** = pokaż użytkownika, jego numer UID oraz grupy, do których należy

□□ **ZMIENIA SYSTEM**

Utworzenie tabeli z kluczem obcym

```
CREATE TABLE zamowienia (
    id SERIAL PRIMARY KEY,
    klient_id INTEGER NOT NULL,
    data_zlozenia DATE NOT NULL DEFAULT CURRENT_DATE,
    FOREIGN KEY (klient_id) REFERENCES klienci(id)
);
```

Kolumna `klient_id` może przyjmować wyłącznie wartości, które istnieją w kolumnie `id` tabeli `klienci` — silnik odrzuci wstawienie zamówienia z nieistniejącym klientem.

Przykład:

```
CREATE TABLE
```

□□ **id** = pokaż użytkownika, jego numer UID oraz grupy, do których należy

□□ **ZMIENIA SYSTEM**

Dodanie klucza obcego do istniejącej tabeli

```
ALTER TABLE zamowienia
    ADD CONSTRAINT fk_zamowienia_klient
    FOREIGN KEY (klient_id) REFERENCES klienci(id);
```

Ten sam efekt na tabeli, która już istnieje i już ma dane — silnik przy wykonaniu tej komendy sprawdza wszystkie istniejące wiersze pod kątem zgodności, zanim ograniczenie zacznie obowiązywać.

Przykład:

```
ALTER TABLE
```

◫◫ TYLKO ODCZYT

Sprawdzenie kluczy tabeli przez information_schema

```
SELECT constraint_name, constraint_type
FROM information_schema.table_constraints
WHERE table_name = 'zamowienia';
```

Przykład:

```
      constraint_name       | constraint_type
----------------------------+-----------------
 zamowienia_pkey             | PRIMARY KEY
 fk_zamowienia_klient        | FOREIGN KEY
(2 rows)
```

Po co to istnieje

Tabela relacyjna, sama z siebie, nie ma pojęcia „ten konkretny wiersz". Ma zbiór kolumn i zbiór wierszy — żeby wskazać jeden, dokładnie jeden wiersz, potrzebny jest sposób na jego jednoznaczne odróżnienie od wszystkich pozostałych. Do tego służy klucz.

> **Pojęcie: klucz.** Minimalny zbiór kolumn, którego wartości jednoznacznie identyfikują wiersz w tabeli — żadne dwa wiersze nie mogą mieć w tych kolumnach tej samej kombinacji wartości. Słowo „minimalny" jest tu istotne: jeśli usunięcie którejkolwiek kolumny ze zbioru sprawia, że jednoznaczność znika, zbiór jest kluczem; jeśli można coś odjąć i jednoznaczność zostaje, to nie jest jeszcze klucz w ścisłym sensie, tylko coś większego.

Rodzaje kluczy

Zanim padnie wyliczenie, warto ustalić, że wszystkie poniższe pojęcia opisują tę samą podstawową ideę — jednoznaczną identyfikację wiersza — różniącą się tym, jak bardzo są minimalne i jaką rolę pełnią w tabeli.

Pojęcie: superklucz. Dowolny zbiór kolumn, którego wartości jednoznacznie identyfikują wiersz — bez wymogu minimalności. Cała tabela (wszystkie jej kolumny razem) zawsze jest superkluczem, bo dwa identyczne wiersze pod każdym względem są w praktyce nieodróżnialne i tak samo traktowane.

Pojęcie: klucz kandydujący. Superklucz, z którego nie da się odjąć żadnej kolumny bez utraty jednoznaczności — czyli superklucz minimalny. Tabela może mieć kilku kandydatów naraz: `klienci` z kolumnami `id`, `nip` i `email` (każda z osobna unikalna) ma trzy klucze kandydujące, jeśli każda kolumna sama w sobie jednoznacznie identyfikuje klienta.

Pojęcie: klucz główny (primary key). Ten jeden klucz kandydujący, który wybieramy jako oficjalny identyfikator wiersza. Silnik dodatkowo wymusza na nim `NOT NULL` — klucz główny nigdy nie może być pusty, bo pusty identyfikator nie identyfikuje niczego.

Pojęcie: klucz obcy (foreign key). Kolumna albo zestaw kolumn w jednej tabeli, którego wartości muszą odpowiadać wartościom klucza (zwykle głównego) w innej tabeli — albo być puste, jeśli kolumna dopuszcza `NULL`. To mechanizm, który fizycznie realizuje relację między tabelami, opisaną na stronie Relacje: 1:1, 1:N, N:M i tabela łącząca.

Pojęcie: klucz unikalny (unique key). Ograniczenie mówiące, że wartości w danej kolumnie (albo zestawie kolumn) się nie powtarzają — bez pretensji do bycia głównym identyfikatorem wiersza. Różni się od klucza głównego dwiema rzeczami: tabela może mieć wiele ograniczeń `UNIQUE` naraz, oraz — w większości silników — kolumna `UNIQUE` dopuszcza wielokrotne wystąpienie `NULL`, bo `NULL` w logice SQL nie jest równy samemu sobie, więc dwie wartości `NULL` nie są traktowane jako powtórzenie.

Pojęcie: klucz złożony (composite key). Klucz zbudowany z więcej niż jednej kolumny — żadna z kolumn osobno nie jest unikalna, ale ich para (albo trójka) już tak. Typowy przykład to tabela łącząca `uzytkownicy_role(uzytkownik_id, rola_id)`, gdzie kluczem głównym jest para tych dwóch kolumn naraz.

Zestawienie wszystkich pięciu obok siebie porządkuje, który termin odpowiada na jakie pytanie:

Rodzaj klucza	Pytanie, na które odpowiada	Ile może ich być w tabeli
Superklucz	jaki zbiór kolumn na pewno identyfikuje wiersz	dowolnie wiele
Klucz kandydujący	jaki minimalny zbiór kolumn identyfikuje wiersz	zero, jeden albo wiele
Klucz główny	który klucz kandydujący wybraliśmy jako oficjalny	dokładnie jeden
Klucz unikalny	jakie inne kolumny też się nie powtarzają	wiele, niezależnie od klucza głównego
Klucz obcy	skąd ta kolumna bierze dopuszczalne wartości	wiele, po jednym na każdą relację z inną tabelą

Naturalny czy sztuczny

Pojęcie: klucz naturalny. Identyfikator zaczerpnięty z danych, które i tak istnieją w świecie rzeczywistym opisywanym przez tabelę — NIP firmy, numer PESEL osoby, adres e-mail konta. Zaletą jest to, że nie wymaga dodatkowej kolumny wyłącznie na potrzeby bazy, ma sens dla człowieka czytającego dane wprost, i naturalnie zapobiega powstaniu dwóch wierszy dla tego samego bytu rzeczywistego, bo sam byt narzuca jedną wartość klucza. Wadą jest to, że dane ze świata rzeczywistego bywają zmienne — adres e-mail się zmienia, numer telefonu bywa przydzielany ponownie po latach, a to, co wyglądało na unikalne (PESEL), potrafi mieć wyjątki (obcokrajowcy bez nadanego numeru, błędy historyczne w danych migrowanych ze starszych systemów). Zmiana wartości klucza naturalnego jest kosztowna, bo trzeba zaktualizować każdy wiersz, który się do niej odwołuje przez klucz obcy.

Pojęcie: klucz sztuczny (surogatowy, surrogate key). Identyfikator wygenerowany przez sam system, bez żadnego znaczenia biznesowego — liczba z sekwencji, `SERIAL`, `IDENTITY`, `AUTO_INCREMENT` albo UUID. Zaletą jest stabilność: wartość nigdy się nie zmienia, bo nic w świecie rzeczywistym jej nie determinuje, złączenia między tabelami są tanie (porównanie liczb całkowitych), a dodanie nowego wiersza nigdy nie koliduje z istniejącym z powodu zmiany danych źródłowych. Wadą jest to, że sam w sobie nic nie mówi o wierszu — dwa wiersze o `id = 41` i `id = 42` mogą przypadkiem opisywać tego samego klienta wpisanego dwa razy, bo klucz sztuczny nie chroni przed duplikatem biznesowym; tę ochronę trzeba dołożyć osobno, zwykle właśnie przez ograniczenie `UNIQUE` na kolumnie z kluczem naturalnym, obok klucza sztucznego jako głównego.

W praktyce większość schematów na tej półce łączy oba podejścia: klucz sztuczny jako klucz główny, bo tani w złączeniach i stabilny, oraz ograniczenie `UNIQUE` na kluczu naturalnym tam, gdzie

taki istnieje, żeby nie stracić ochrony przed duplikatem.

Klucz obcy może wskazywać też klucz unikalny, nie tylko główny

Klucz obcy nie musi odwoływać się wyłącznie do klucza głównego tabeli nadrzędnej — wystarczy, że wskazuje kolumnę objętą jakimkolwiek ograniczeniem gwarantującym jednoznaczność, czyli kluczem głównym albo kluczem unikalnym. Jeśli tabela `klienci` ma klucz główny `id`, ale też unikalny `nip`, to klucz obcy w innej tabeli może równie dobrze odwoływać się do `nip`, o ile intencją jest powiązanie po numerze identyfikacji podatkowej, a nie po sztucznym identyfikatorze. Silnik i tak wymaga, żeby kolumna, do której odwołuje się klucz obcy, była objęta ograniczeniem unikalności — bez tego nie dałoby się jednoznacznie rozstrzygnąć, do którego wiersza tabeli nadrzędnej odwołanie się odnosi.

Jak to działa

Większość silników zakłada indeks pod kluczem głównym automatycznie — w PostgreSQL, MSSQL i MySQL `PRIMARY KEY` tworzy unikalny indeks B-drzewa bez dodatkowej komendy. Klucz obcy takiego automatycznego indeksu w PostgreSQL i MSSQL nie dostaje — kolumna `klient_id` w tabeli `zamowienia` jest sprawdzana pod kątem istnienia wartości w tabeli `klienci`, ale sama w sobie nie jest indeksowana, dopóki nie założymy indeksu ręcznie. MySQL z silnikiem InnoDB robi to inaczej i zakłada indeks pod kolumną klucza obcego automatycznie.

Sprawdzenie klucza obcego wykonuje się przy każdym `INSERT` i `UPDATE` na tabeli, która go posiada, oraz przy każdym `DELETE` na tabeli, do której się odwołuje — silnik musi zweryfikować, że usunięcie wiersza nadrzędnego nie zostawi osieroconych odwołań. Domyślne zachowanie w takiej sytuacji to zablokowanie usunięcia (`ON DELETE RESTRICT` albo `NO ACTION`, zależnie od silnika); można je zmienić na kaskadowe usunięcie zależnych wierszy (`ON DELETE CASCADE`) albo ustawienie klucza obcego na `NULL` (`ON DELETE SET NULL`), jeśli kolumna na to pozwala.

Częste błędy

Brak indeksu na kolumnie klucza obcego w PostgreSQL. Wielu programistów zakłada, że skoro `FOREIGN KEY` istnieje, kolumna jest zindeksowana. W PostgreSQL i MSSQL nie jest — każde usunięcie wiersza z tabeli nadrzędnej wymusza pełny skan tabeli podrzędnej w poszukiwaniu wierszy zależnych, co przy dużej tabeli oznacza zauważalne spowolnienie i długo trzymaną blokadę.

Traktowanie `UNIQUE` jak lżejszej wersji `PRIMARY KEY`. Różnica nie jest tylko kosmetyczna. Kolumna z ograniczeniem `UNIQUE` pozwala na wielokrotne `NULL`, więc próba użycia jej jako jedynego zabezpieczenia przed duplikatami bez dodatkowego `NOT NULL` nie zadziała tak, jak ktoś by oczekiwał — dwa wiersze z `NULL` w tej kolumnie przejdą bez błędu.

Traktowanie klucza naturalnego jako niezmiennego, bo tak wygląda na pierwszy rzut oka. Numer PESEL jest jedyny w swoim rodzaju dla obywatela, ale system, który migrował dane historyczne, ma w bazie klientów zagranicznych bez nadanego numeru albo z numerem tymczasowym, który później się zmienił. Jeśli PESEL jest kluczem głównym, a nie tylko atrybutem z ograniczeniem `UNIQUE`, każda taka zmiana wymusza aktualizację wszystkich wierszy, które się do niego odwołują przez klucz obcy — dokładnie tego problemu unika klucz sztuczny.

Dodanie klucza obcego do tabeli, która referuje jeszcze nieistniejącą tabelę. Komenda `CREATE TABLE zamowienia (... FOREIGN KEY (klient_id) REFERENCES klienci(id))` zwraca błąd, jeśli `klienci` jeszcze nie istnieje — silnik nie potrafi sprawdzić klucza obcego wobec czegoś, czego nie ma. Kolejność tworzenia tabel musi iść od tabeli nadrzędnej do podrzędnej, albo klucz obcy trzeba dodać później przez `ALTER TABLE`.

□□ Ćwiczenia

> Poniższe zadania wykonujemy na dowolnym silniku SQL z uprawnieniami do tworzenia tabel we własnej, testowej bazie.

Ćwiczenie 1 — złamanie klucza obcego

```
CREATE TABLE klienci (id SERIAL PRIMARY KEY, nazwa TEXT);
CREATE TABLE zamowienia (
    id SERIAL PRIMARY KEY,
    klient_id INTEGER REFERENCES klienci(id)
);
INSERT INTO zamowienia (klient_id) VALUES (999);
```

Co zwróci silnik i dlaczego?

Odpowiedź

Błąd naruszenia klucza obcego, bo w tabeli `klienci` nie ma jeszcze wiersza z `id = 999`:

```
ERROR:  insert or update on table "zamowienia" violates foreign key constraint
DETAIL:  Key (klient_id)=(999) is not present in table "klienci".
```

Ćwiczenie 2 — klucz naturalny kontra sztuczny

Projektujemy tabelę `pracownicy` w firmie, która zatrudnia zarówno obywateli z nadanym numerem PESEL, jak i pracowników zagranicznych bez niego. Który klucz wybieramy jako główny i co robimy z PESEL-em?

Odpowiedź

Kluczem głównym zostaje kolumna sztuczna, np. `id SERIAL PRIMARY KEY` — działa identycznie dla każdego pracownika, niezależnie od tego, czy ma PESEL. Kolumna `pesel` zostaje jako zwykły atrybut z ograniczeniem `UNIQUE`, które dopuszcza `NULL` dla pracowników bez numeru (w większości silników wiele wartości `NULL` w kolumnie `UNIQUE` nie koliduje ze sobą). Dzięki temu baza wciąż pilnuje, żeby dwóch różnych pracowników nie miało tego samego PESEL-u, nie wymuszając go tam, gdzie go po prostu nie ma.

Pytania kontrolne

1. Tabela `produkty` ma kolumny `id` (sztuczny klucz główny) i `kod_kreskowy` (unikalny w świecie rzeczywistym, ale bez ograniczenia `UNIQUE` w bazie). Co może pójść źle?

Odpowiedź

Nic w bazie nie broni wstawienia dwóch produktów z tym samym kodem kreskowym — każdy dostanie inny `id`, więc klucz główny się nie poskarży, a duplikat biznesowy powstanie po cichu. Naprawa to dodanie `ALTER TABLE produkty ADD CONSTRAINT uq_produkty_kod UNIQUE (kod_kreskowy);`.

2. Dlaczego klucz złożony `(uzytkownik_id, rola_id)` jako klucz główny tabeli łączącej jest lepszym rozwiązaniem niż dodanie do niej osobnej, sztucznej kolumny `id`?

Odpowiedź

Bo sens tabeli łączącej to właśnie ta para — jeden użytkownik z jedną rolą występuje najwyżej raz. Klucz złożony wymusza tę regułę za darmo, przy okazji bycia kluczem głównym. Dodatkowa kolumna `id` niczego by nie dodała poza koniecznością osobnego ograniczenia `UNIQUE (uzytkownik_id, rola_id)`, żeby uzyskać tę samą gwarancję, którą klucz złożony daje wprost.

Powiązane: Model relacyjny: tabela, wiersz, kolumna, dziedzina · Relacje: 1:1, 1:N, N:M i tabela łącząca · Ograniczenia: NOT NULL, UNIQUE, CHECK, FOREIGN KEY, DEFAULT · Sekwencje, IDENTITY, AUTO_INCREMENT

Źródła: dokumentacja PostgreSQL, rozdz. „Constraints" · C. J. Date, „An Introduction to Database Systems", rozdz. o kluczach kandydujących

Relacje: 1:1, 1:N, N:M i tabela łącząca

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

1:1, 1:N i N:M opisują, ile wierszy jednej tabeli może odpowiadać ilu wierszom drugiej — i tylko jedna z tych trzech kardynalności wymaga trzeciej tabeli, żeby w ogóle dało się ją zapisać.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

ZMIENIA SYSTEM

Realizacja relacji 1:N kluczem obcym po stronie „wielu"

```
CREATE TABLE pozycje_zamowienia (
    id SERIAL PRIMARY KEY,
    zamowienie_id INTEGER NOT NULL REFERENCES zamowienia(id),
    produkt_id INTEGER NOT NULL,
    ilosc INTEGER NOT NULL CHECK (ilosc > 0)
);
```

Klucz obcy `zamowienie_id` siedzi po stronie tabeli, która ma być „wieloma" — jedno zamówienie może mieć wiele pozycji, ale każda pozycja wskazuje dokładnie jedno zamówienie.

Przykład:

```
CREATE TABLE
```

□□ **id** = pokaż użytkownika, jego numer UID oraz grupy, do których należy

□□ ZMIENIA SYSTEM

Realizacja relacji N:M tabelą łączącą

```
CREATE TABLE uzytkownicy_role (
    uzytkownik_id INTEGER NOT NULL REFERENCES uzytkownicy(id),
    rola_id INTEGER NOT NULL REFERENCES role(id),
    PRIMARY KEY (uzytkownik_id, rola_id)
);
```

Żadna z tabel `uzytkownicy` ani `role` nie dostaje klucza obcego do drugiej — relacja N:M mieszka wyłącznie w tabeli pośredniej.

Przykład:

```
CREATE TABLE
```

Po co to istnieje

Na stronie Model relacyjny: tabela, wiersz, kolumna, dziedzina słowo „relacja" oznaczało tabelę. Tutaj to samo słowo pojawia się w drugim, potocznym znaczeniu — jako powiązanie między dwiema tabelami: klient ma zamówienia, zamówienie ma pozycje, użytkownik ma role. Oba znaczenia funkcjonują w rozmowach o bazach danych równolegle, i kontekst zwykle wystarcza, żeby się nie pomylić, ale warto mieć to rozróżnienie nazwane wprost, zanim padnie reszta tej strony.

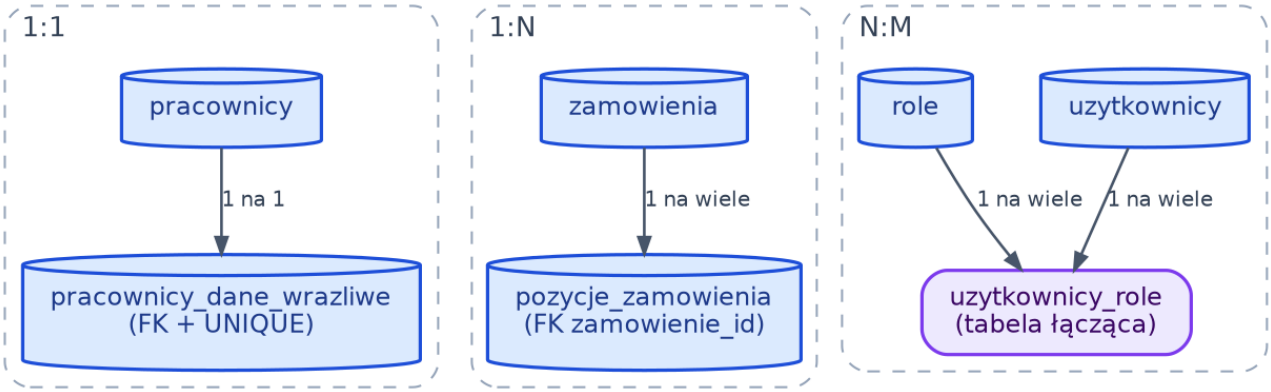
> **Pojęcie: relacja między tabelami.** Powiązanie logiczne, które mówi, jak wiersze jednej tabeli odnoszą się do wierszy drugiej. Kluczowa własność takiej relacji to jej kardynalność — czyli ile wierszy po jednej stronie może odpowiadać ilu wierszom po drugiej. Kardynalności są trzy: jeden do jednego, jeden do wielu i wiele do wielu, i to one — nie technika ich zapisu — decydują, jak relację zaprojektować.

Analogia

Wyobraźmy sobie szkołę. Uczeń i jego numer legitymacji szkolnej są w relacji 1:1 — jeden uczeń ma dokładnie jedną legitymację, a jedna legitymacja należy do dokładnie jednego ucznia; gdyby ktoś zgubił legitymację, dostaje nową o innym numerze, a nie cudzą. Uczeń i klasa, do której jest przypisany na stałe, to relacja 1:N — jedna klasa ma wielu uczniów, ale każdy uczeń w danym roku szkolnym należy do dokładnie jednej klasy. Uczeń i przedmioty, na które uczęszcza, to relacja N:M — jeden uczeń chodzi na wiele przedmiotów, a na jeden przedmiot (matematykę) chodzi wielu uczniów naraz.

Analogia pokazuje też, dlaczego N:M potrzebuje osobnego bytu. Sekretariat szkoły nie prowadzi tej relacji, dopisując w kartotece ucznia listę przedmiotów ani w kartotece przedmiotu listę uczniów — prowadzi osobny rejestr zapisów, gdzie każdy wiersz to jedno zdarzenie: ten uczeń zapisany na ten przedmiot w tym roku. Ten rejestr jest dokładnie tym, czym w bazie danych jest tabela łącząca.

Trzy kardynalności



Kardynalność 1:1, 1:N i N:M: dwie tabele połączone kluczem obcym, a przy N:M trzecia tabela łącząca pośrodku (kliknięcie otwiera pełny rozmiar)

1:1 — jeden do jednego. Jeden wiersz tabeli A odpowiada najwyżej jednemu wierszowi tabeli B, i odwrotnie. W praktyce spotyka się to rzadziej niż pozostałe dwie kardynalności, zwykle jako świadomy podział jednej logicznej encji na dwie tabele — na przykład `pracownicy` (dane podstawowe, czytane często) i `pracownicy_dane_wrazliwe` (numer konta, dane medyczne, czytane rzadko i z węższymi uprawnieniami). Realizuje się to kluczem obcym, na którym dodatkowo nałożone jest ograniczenie `UNIQUE` — bez `UNIQUE` kolumna klucza obcego pozwoliłaby wielu wierszom tabeli B wskazywać ten sam wiersz tabeli A, co przeistoczyłoby relację w 1:N.

1:N — jeden do wielu. Jeden wiersz tabeli A odpowiada wielu wierszom tabeli B, ale każdy wiersz tabeli B odpowiada dokładnie jednemu wierszowi tabeli A. To najczęstsza kardynalność w praktycznych schematach. Klasyczny przykład: jedno zamówienie (`zamowienia`) ma wiele pozycji (

`pozycje_zamowienia`), ale każda pozycja należy do dokładnie jednego zamówienia. Realizuje się ją kluczem obcym umieszczonym po stronie „wielu" — to `pozycje_zamowienia` trzyma kolumnę `zamowienie_id`, nie odwrotnie, bo gdyby `zamowienia` miały kolumnę wskazującą pozycję, jeden wiersz mógłby wskazać tylko jedną pozycję, co przeczyłoby idei zamówienia z wieloma pozycjami.

N:M — wiele do wielu. Wiele wierszy tabeli A odpowiada wielu wierszom tabeli B, i odwrotnie. Przykład: użytkownik może mieć wiele ról (administrator i redaktor naraz), a jedna rola (administrator) przysługuje wielu użytkownikom. Tej kardynalności nie da się zapisać kluczem obcym w żadnej z dwóch tabel wprost — kolumna w relacyjnej tabeli przechowuje jedną wartość, nie listę wartości, więc `uzytkownicy.rola_id` mogłoby wskazać tylko jedną rolę na użytkownika, co jest sprzeczne z założeniem. Rozwiązaniem jest trzecia tabela.

Dlaczego N:M zawsze potrzebuje tabeli łączącej

> **Pojęcie: tabela łącząca (pośrednia).** Tabela, której jedynym zadaniem jest przechowywanie par kluczy obcych wskazujących po jednym wierszu z każdej z dwóch tabel biorących udział w relacji N:M. Każdy wiersz tabeli łączącej reprezentuje jedno konkretne skojarzenie — ten użytkownik ma tę rolę.

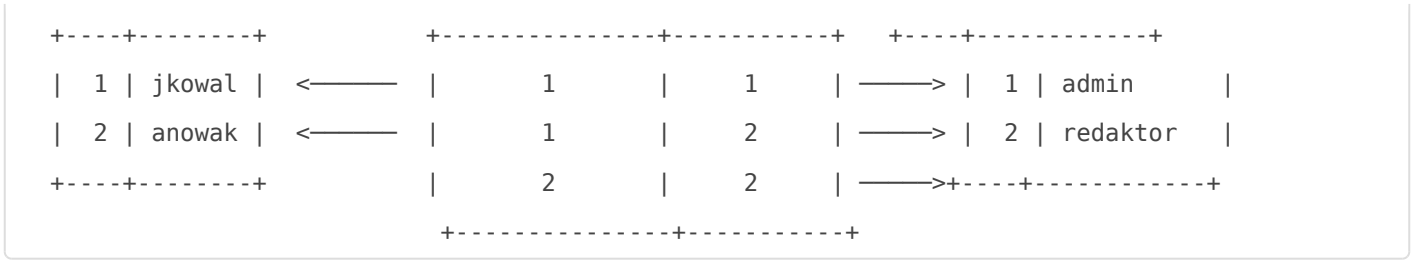
Powód, dla którego to jedyne rozwiązanie w modelu relacyjnym, wynika wprost z zasady, że kolumna tabeli przechowuje pojedynczą, atomową wartość — nie tablicę ani listę. Rozbicie relacji N:M na dwie relacje 1:N sprowadza problem do czegoś, co już umiemy zapisać: tabela łącząca jest w relacji 1:N zarówno z tabelą `uzytkownicy` (jeden użytkownik, wiele wierszy w tabeli łączącej), jak i z tabelą `role` (jedna rola, wiele wierszy w tabeli łączącej). Klucz główny tabeli łączącej jest zwykle złożony z obu kluczy obcych naraz (`PRIMARY KEY (uzytkownik_id, rola_id)`), co dodatkowo wymusza, że dana para użytkownik-rola nie powtórzy się dwa razy.

```
zamowienia (1) ------------------ (N) pozycje_zamowienia
    id  <──────────────────────────────  zamowienie_id



uzytkownicy (N) --- (N) role
                przez tabelę łączącą:

  uzytkownicy                uzytkownicy_role                role
 +----+--------+           +---------------+-----------+   +----+------------+
 | id | login  |           | uzytkownik_id | rola_id   |   | id | nazwa      |
```



Jak to działa

Silnik nie ma osobnego mechanizmu dla „relacji N:M" — dla niego istnieją wyłącznie tabele i klucze obce, a N:M to wzorzec zbudowany z dwóch relacji 1:N. Zapytanie, które chce zobaczyć role danego użytkownika, musi więc przejść przez tabelę pośrednią złączeniem — nie da się dostać z `uzytkownicy` prosto do `role` bez przejścia przez `uzytkownicy_role`:

```
SELECT r.nazwa
FROM role r
JOIN uzytkownicy_role ur ON ur.rola_id = r.id
WHERE ur.uzytkownik_id = 1;
```

Ta sama mechanika kluczy obcych, która wymusza istnienie tabeli nadrzędnej przy 1:N, działa też w tabeli łączącej — próba wstawienia pary z nieistniejącym `uzytkownik_id` albo `rola_id` zostanie odrzucona przez oba ograniczenia `FOREIGN KEY` naraz.

Relacja 1:1 bywa czasem realizowana jeszcze inaczej — zamiast osobnej kolumny z `UNIQUE`, obie tabele dzielą ten sam klucz główny (klucz główny tabeli B jest jednocześnie kluczem obcym do tabeli A). Wtedy `pracownicy_dane_wrazliwe.id` jest zarówno kluczem głównym tej tabeli, jak i kluczem obcym do `pracownicy.id` — co dodatkowo gwarantuje relację 1:1 bez osobnego ograniczenia `UNIQUE`, bo klucz główny z definicji już jest unikalny.

Usuwanie wierszy powiązanych relacją

Skasowanie wiersza, który jest stroną „jeden" w relacji 1:N albo jednym z uczestników relacji N:M, rodzi pytanie, co ma się stać z wierszami, które się do niego odwołują. Silnik pozwala to zdefiniować przy tworzeniu klucza obcego przez klauzulę `ON DELETE`.

Domyślne zachowanie w większości silników to odrzucenie usunięcia (`RESTRICT` albo `NO ACTION`) — nie da się skasować klienta, dopóki istnieją jego zamówienia. `ON DELETE CASCADE` usuwa razem z wierszem nadrzędnym wszystkie wiersze podrzędne — skasowanie zamówienia automatycznie

kasuje jego pozycje, co dla relacji 1:N między `zamowienia` i `pozycje_zamowienia` zwykle jest właściwym zachowaniem, bo pozycja bez zamówienia nie ma sensu. `ON DELETE SET NULL` zostawia wiersz podrzędny, ale zeruje klucz obcy — sensowne tylko wtedy, gdy kolumna klucza obcego dopuszcza `NULL` i gdy istnienie wiersza podrzędnego bez rodzica ma jakiekolwiek znaczenie biznesowe.

W tabeli łączącej N:M niemal zawsze stosuje się `ON DELETE CASCADE` po obu stronach — usunięcie użytkownika powinno usunąć jego przypisania do ról, nie zostawiać w `uzytkownicy_role` osierocone wiersze wskazujące nieistniejącego już użytkownika.

Częste błędy

Modelowanie N:M listą wartości w jednej kolumnie. Kolumna `uzytkownicy.role` z wartością `'admin,redaktor'` technicznie działa — silnik przyjmie taki tekst bez sprzeciwu — ale łamie założenie, że kolumna trzyma pojedynczą, atomową wartość. Wyszukanie wszystkich administratorów wymaga wtedy dopasowania podciągu tekstu zamiast prostego złączenia, dodanie roli wymaga edycji tekstu zamiast wstawienia wiersza, a nic nie broni literówki w nazwie roli w środku takiego ciągu.

Brak złożonego klucza głównego na tabeli łączącej. Tabela `uzytkownicy_role` z osobną kolumną `id` jako jedynym kluczem głównym, bez ograniczenia `UNIQUE (uzytkownik_id, rola_id)`, dopuszcza wstawienie tej samej pary użytkownik-rola wielokrotnie — z pozoru nieszkodliwe, dopóki raport licząc przypisania ról nie zacznie liczyć duplikatów.

Modelowanie 1:1 dwoma niezależnymi kluczami obcymi bez `UNIQUE`. Dwie tabele połączone zwykłym kluczem obcym bez dodatkowego ograniczenia unikalności są w relacji 1:N, nie 1:1 — nic nie stoi na przeszkodzie, żeby dwa wiersze tabeli B wskazywały ten sam wiersz tabeli A. Jeśli zamiarem było 1:1, taki brak ograniczenia jest cichym błędem projektowym, który ujawni się dopiero jako duplikat w produkcji.

□□ Ćwiczenia

> Poniższe zadania wykonujemy na dowolnym silniku SQL z uprawnieniami do tworzenia tabel we własnej, testowej bazie.

Ćwiczenie 1 — zaprojektowanie relacji

Mamy zaprojektować schemat dla biblioteki: książki, czytelnicy, wypożyczenia. Jeden czytelnik może wypożyczyć wiele książek w różnym czasie, a jedna książka (jeden konkretny egzemplarz) w

danym momencie może być wypożyczona przez jednego czytelnika, ale w historii — przez wielu, jeden po drugim. Jaka to kardynalność między `czytelnicy` i `ksiazki`, i gdzie ląduje klucz obcy?

Odpowiedź

To 1:N po obu stronach naraz, rozwiązywane trzecią tabelą `wypozyczenia` — nie dlatego, że jeden czytelnik może mieć wiele wypożyczonych książek w danym momencie (to samo w sobie dałoby się zapisać kluczem obcym w `wypozyczenia.czytelnik_id`), ale dlatego, że jedna książka jest wypożyczana wielokrotnie w czasie przez różnych czytelników — historia wypożyczeń wymaga osobnego wiersza na każde zdarzenie, nie da się jej zapisać jako pojedynczej kolumny w `ksiazki`. Tabela `wypozyczenia` ma kolumny `czytelnik_id`, `ksiazka_id`, `data_wypozyczenia`, `data_zwrotu` — i to jest wzorcowa tabela łącząca, tyle że z dodatkowymi kolumnami opisującymi samo zdarzenie, nie tylko parę kluczy.

Pytania kontrolne

1. Dlaczego relacji N:M nie da się zapisać, dodając do tabeli `uzytkownicy` kolumnę `rola_id` wskazującą tabelę `role`?

Odpowiedź

Bo kolumna `rola_id` w jednym wierszu trzyma dokładnie jedną wartość, a N:M zakłada, że jeden użytkownik może mieć wiele ról naraz. Taka kolumna zrealizowałaby co najwyżej relację N:1 (wielu użytkowników, jedna rola każdy), nie N:M.

2. Tabela łącząca `uzytkownicy_role` dostała dodatkową kolumnę `nadano_od DATE`. Czy to coś zmienia w jej roli jako tabeli łączącej?

Odpowiedź

Nie zmienia jej roli — tabela łącząca może przechowywać dodatkowe atrybuty samej relacji (kiedy rola została nadana, kto ją nadał), o ile nie łamie to klucza głównego złożonego z obu kluczy obcych. To bardzo częsty i poprawny wzorzec: relacja N:M z własnymi cechami, a nie tylko gołą parą identyfikatorów.

Powiązane: Model relacyjny: tabela, wiersz, kolumna, dziedzina · Klucze: główny, obcy, unikalny, naturalny, sztuczny · Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza · Normalizacja do 3NF i świadoma denormalizacja

Źródła: C. J. Date, „An Introduction to Database Systems", rozdz. o związkach między relacjami · dokumentacja PostgreSQL, rozdz. „Foreign Keys"

Katalog systemowy: skąd baza wie, co w sobie ma

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~20 min**

Baza danych wie, jakie ma tabele i kolumny, bo ta informacja też jest zapisana w tabelach — tyle że w tabelach należących do samej bazy, nie do nas.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
ANSI	American National Standards Institute: tu chodzi o kody sterujące kolorem i pozycją kursora w terminalu
DDL	Data Definition Language: polecenia SQL zmieniające strukturę (CREATE, ALTER, DROP)
MSSQL	Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

TYLKO ODCZYT

Wylistowanie tabel przez information_schema (standard ANSI)

```
SELECT table_schema, table_name
FROM information_schema.tables
WHERE table_type = 'BASE TABLE'
ORDER BY table_schema, table_name;
```

Ten sam zapis działa bez zmian w PostgreSQL, MySQL i MSSQL, bo `information_schema` jest częścią standardu SQL, nie wynalazkiem jednego silnika.

Przykład (PostgreSQL, psql):

```
 table_schema |    table_name
--------------+------------------
 public       | klienci
 public       | pozycje_zamowienia
 public       | zamowienia
(3 rows)
```

ANSI = American National Standards Institute: tu chodzi o kody sterujące kolorem i pozycją kursora w terminalu · **MSSQL** = Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima · **SQL** = Structured Query Language: język zapytań do relacyjnych baz danych

TYLKO ODCZYT

Wylistowanie kolumn tabeli przez information_schema

```
SELECT column_name, data_type, is_nullable
FROM information_schema.columns
WHERE table_name = 'zamowienia'
ORDER BY ordinal_position;
```

Przykład:

```
 column_name   |     data_type     | is_nullable
---------------+-------------------+-------------
 id            | integer           | NO
 klient_id     | integer           | NO
 data_zlozenia | date              | NO
(3 rows)
```

TYLKO ODCZYT

Wylistowanie tabel przez katalog natywny PostgreSQL: pg_class

```
SELECT relname
FROM pg_catalog.pg_class
WHERE relkind = 'r'
  AND relnamespace = 'public'::regnamespace
ORDER BY relname;
```

`pg_class` to jedna z centralnych tabel katalogu PostgreSQL — trzyma po jednym wierszu na każdy obiekt „podobny do tabeli": tabelę, widok, indeks, sekwencję. Kolumna `relkind` odróżnia je od siebie (`r` to zwykła tabela).

Przykład:

```
       relname
---------------------
 klienci
 pozycje_zamowienia
 zamowienia
(3 rows)
```

ZMIENIA SYSTEM

Wylistowanie kolumn tabeli przez katalog natywny PostgreSQL: pg_attribute

```
SELECT attname, atttypid::regtype
FROM pg_catalog.pg_attribute
WHERE attrelid = 'zamowienia'::regclass
  AND attnum > 0
  AND NOT attisdropped
ORDER BY attnum;
```

Warunek `attnum > 0` odsiewa kolumny systemowe (`ctid`, `xmin`), a `NOT attisdropped` odsiewa kolumny formalnie usunięte, ale wciąż fizycznie zajmujące miejsce w definicji wiersza.

Przykład:

```
    attname     | atttypid
----------------+-----------
 id             | integer
 klient_id      | integer
 data_zlozenia  | date
(3 rows)
```

TYLKO ODCZYT

Wylistowanie tabel w Oracle: ALL_TABLES

```
SELECT owner, table_name
FROM all_tables
WHERE owner = 'SKLEP'
ORDER BY table_name;
```

Przykład (sqlplus):

```
OWNER      TABLE_NAME
---------- ------------------------------
SKLEP      KLIENCI
SKLEP      POZYCJE_ZAMOWIENIA
SKLEP      ZAMOWIENIA

3 rows selected.
```

TYLKO ODCZYT

Wylistowanie kolumn tabeli w Oracle: USER_TAB_COLUMNS

```
SELECT column_name, data_type, nullable
FROM user_tab_columns
WHERE table_name = 'ZAMOWIENIA'
ORDER BY column_id;
```

`USER_TAB_COLUMNS` pokazuje wyłącznie obiekty należące do schematu, którym właśnie jesteśmy zalogowani — bez potrzeby filtrowania po właścicielu, w przeciwieństwie do `ALL_TAB_COLUMNS`.

Przykład:

```
COLUMN_NAME        DATA_TYPE       N
------------------ --------------- -
ID                  NUMBER          N
KLIENT_ID           NUMBER          N
DATA_ZLOZENIA       DATE            N

3 rows selected.
```

TYLKO ODCZYT

Wylistowanie tabel w MSSQL: sys.tables

```
SELECT SCHEMA_NAME(schema_id) AS schemat, name
FROM sys.tables
ORDER BY schemat, name;
```

Przykład (sqlcmd):

```
schemat      name
------------ --------------------
dbo          Klienci
dbo          PozycjeZamowienia
dbo          Zamowienia

(3 rows affected)
```

MSSQL = Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima

TYLKO ODCZYT

Wylistowanie kolumn tabeli w MSSQL: sys.columns

```
SELECT name, TYPE_NAME(system_type_id) AS typ, is_nullable
FROM sys.columns
WHERE object_id = OBJECT_ID('dbo.Zamowienia')
ORDER BY column_id;
```

Przykład:

```
name            typ        is_nullable
--------------- ---------- -----------
Id              int        0
KlientId        int        0
DataZlozenia    date       0

(3 rows affected)
```

MSSQL = Microsoft SQL Server: silnik bazy danych Microsoftu, na nim w Zielonej stoi księgowość Optima

Po co to istnieje

Zapytanie `SELECT * FROM zamowienia` zwraca dane z tabeli `zamowienia`, ale ktoś — albo coś — musiało wcześniej wiedzieć, że taka tabela w ogóle istnieje, ile ma kolumn i jakiego są typu, zanim silnik zdołał ten `SELECT` w ogóle sparsować. Ta wiedza nie jest wbudowana w kod silnika na sztywno — jest zapisana w bazie, obok naszych własnych danych, w postaci zwykłych tabel.

> **Pojęcie: katalog systemowy (słownik danych).** Zbiór tabel i widoków, które opisują strukturę samej bazy — jakie ma tabele, kolumny, indeksy, użytkowników, uprawnienia. To dane o danych: metadane. Katalog systemowy jest tworzony i aktualizowany automatycznie przy każdej komendzie `CREATE`, `ALTER` czy `DROP` — nie edytujemy go bezpośrednio, tylko przez zwykłe polecenia DDL, ale czytać go możemy zwykłym `SELECT`, dokładnie tak jak każdą inną tabelę.

Każdy z czterech głównych silników SQL na tej półce ma dwie równoległe drogi dostępu do tej samej informacji: standard ANSI (`information_schema`) i katalog natywny, specyficzny dla silnika.

> **Pojęcie: information_schema.** Zestaw widoków zdefiniowany w standardzie SQL, jednakowo nazwany i jednakowo ustrukturyzowany w każdym silniku, który go implementuje — PostgreSQL, MySQL i MSSQL. Zaletą jest przenośność: to samo zapytanie o listę tabel działa bez zmian na trzech różnych silnikach. Wadą jest to, że standard opisuje wspólny mianownik, więc szczegóły specyficzne dla danego silnika — fizyczny rozmiar tabeli na dysku, typ metody dostępu indeksu, statystyki optymalizatora — w `information_schema` po prostu nie istnieją.
>
> **Pojęcie: katalog natywny.** Własny, wewnętrzny zestaw tabel systemowych danego silnika, nieopisany żadnym zewnętrznym standardem — `pg_catalog` w PostgreSQL, `ALL_`/`USER_`/`DBA_` w Oracle, schemat `sys` w MSSQL. Daje dostęp do wszystkiego, co silnik o sobie wie, łącznie ze szczegółami niedostępnymi przez `information_schema`, kosztem przenośności — zapytanie do `pg_catalog` nie zadziała na Oracle.

Oracle nie implementuje `information_schema` w ogóle — jedyną drogą jest tam katalog natywny, z rozróżnieniem na trzy rodziny widoków: `USER_*` (obiekty własne bieżącego schematu), `ALL_*` (obiekty własne plus te, do których mamy nadane uprawnienia) i `DBA_*` (wszystkie obiekty w całej bazie, wymaga uprawnień administracyjnych).

Jak to działa

Katalog systemowy to nie magiczna warstwa poza bazą — to zwykłe tabele, żyjące w tej samej bazie co nasze dane, tyle że należące do silnika, a nie do nas. Kiedy wykonujemy `CREATE TABLE zamowienia (...)`, silnik robi dwie rzeczy: zakłada fizyczne miejsce na dane nowej tabeli oraz wstawia wiersze opisujące tę tabelę do własnych tabel katalogowych — w PostgreSQL nowy wiersz ląduje w `pg_class`, a po jednym wierszu na kolumnę w `pg_attribute`. `DROP TABLE` usuwa te wiersze z powrotem. To dlatego zapytanie do katalogu jest zwykłym `SELECT`, a nie osobną komendą — bo katalog jest tabelą jak każda inna, tyle że jej zawartość aktualizuje sam silnik przy okazji DDL.

Widoki `information_schema` w praktyce są zbudowane właśnie nad katalogiem natywnym — w PostgreSQL `information_schema.tables` jest widokiem, który pod spodem odpytuje `pg_class` i `pg_namespace`, tylko w formacie zgodnym ze standardem. To dlatego zapytanie do `information_schema` bywa odrobinę wolniejsze niż to samo pytanie zadane wprost do katalogu natywnego — przechodzi przez dodatkową warstwę widoku.

Częste błędy

Zapytanie do `information_schema.columns` bez filtra na schemat. W bazie, która ma wiele schematów (typowe w PostgreSQL i MSSQL), zapytanie `WHERE table_name = 'zamowienia'` bez dodatkowego `AND table_schema = 'public'` zwróci po jednym wierszu z każdego schematu, w którym istnieje tabela o tej nazwie — myląco sugerując więcej kolumn niż faktycznie ma nasza konkretna tabela.

Filtrowanie po nazwie z niewłaściwą wielkością liter. PostgreSQL składuje niecytowane nazwy identyfikatorów małymi literami — `CREATE TABLE Zamowienia` tworzy w rzeczywistości tabelę `zamowienia`, więc `WHERE table_name = 'Zamowienia'` w `information_schema.tables` nie zwróci nic. Oracle robi odwrotnie: niecytowane identyfikatory zapisuje wielkimi literami, więc tam trzeba filtrować `WHERE table_name = 'ZAMOWIENIA'`. Ten sam nawyk przeniesiony między silnikami gwarantuje pusty wynik.

Odpytywanie `ALL_TAB_COLUMNS` zamiast `USER_TAB_COLUMNS` na dużej, wielo-schematowej bazie Oracle. `ALL_TAB_COLUMNS` przegląda wszystkie obiekty, do których mamy jakiekolwiek uprawnienia w całej bazie — na środowisku z dziesiątkami schematów to zapytanie bywa zauważalnie wolniejsze niż to samo pytanie zawężone przez `USER_TAB_COLUMNS` do własnego schematu.

□□ Ćwiczenia

>

> Poniższe zadania wykonujemy na dowolnym silniku SQL, do którego mamy dostęp z prawem odczytu katalogu.

Ćwiczenie 1 — porównanie dwóch dróg do tej samej informacji

Na PostgreSQL wykonujemy oba zapytania o listę kolumn tabeli `zamowienia` — jedno przez `information_schema.columns`, drugie przez `pg_catalog.pg_attribute` — i porównujemy liczbę zwróconych wierszy.

Odpowiedź

Bez dodatkowych warunków `pg_attribute` może zwrócić więcej wierszy niż `information_schema.columns`, jeśli tabela ma kolumny formalnie usunięte przez `ALTER TABLE ... DROP COLUMN` — PostgreSQL fizycznie nie usuwa takiej kolumny od razu, tylko oznacza ją jako `attisdropped = true` i ukrywa z widoków standardowych. `information_schema.columns` nigdy takich kolumn nie pokazuje; `pg_attribute` pokaże je, jeśli nie dodamy warunku `NOT attisdropped`.

Pytania kontrolne

1. Dlaczego zapytanie do `information_schema.tables` napisane raz działa bez zmian na PostgreSQL, MySQL i MSSQL, ale nie na Oracle?

Odpowiedź

Bo `information_schema` jest częścią standardu SQL i te trzy silniki go implementują z tą samą strukturą widoków. Oracle nigdy nie zaimplementował `information_schema` — jego jedyną drogą dostępu do metadanych jest katalog natywny (`ALL_TABLES`, `USER_TABLES`, `DBA_TABLES`), więc zapytanie napisane pod standard trzeba tam przepisać.

2. Czym różni się `USER_TABLES` od `ALL_TABLES` w Oracle?

Odpowiedź

`USER_TABLES` pokazuje wyłącznie tabele należące do schematu, którym jesteśmy aktualnie zalogowani. `ALL_TABLES` pokazuje te same tabele plus wszystkie inne, do których nasz

użytkownik ma nadane jakiekolwiek uprawnienie (np. przez `GRANT SELECT`), niezależnie od tego, w czyim są schemacie.

Powiązane: Baza danych, silnik, instancja, schemat: cztery pojęcia, które się myli · Klucze: główny, obcy, unikalny, naturalny, sztuczny · Konta, role i uprawnienia: GRANT, REVOKE, zasada najmniejszych uprawnień

Źródła: dokumentacja PostgreSQL, rozdz. „The Information Schema" i „System Catalogs" · Oracle Database Reference, rozdz. „Static Data Dictionary Views" · Microsoft Docs, „Catalog Views (Transact-SQL)"

Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin

Poziom: podstawy **Dotyczy: wszystkie silniki** **Czas: ~15 min**

„NoSQL" sugeruje jedną kategorię, a w rzeczywistości kryje pod tą nazwą trzy albo cztery zupełnie różne sposoby organizowania danych, które łączy właściwie tylko to, że żaden z nich nie jest modelem relacyjnym.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
ASE	Adaptive Server Enterprise: nazwa silnika Sybase od 1996 roku, dziś rozwijanego przez SAP jako SAP ASE
BSON	Binary JSON: binarny format zapisu dokumentów używany wewnętrznie przez MongoDB, w tym w archiwach mongodump
CAP	Client Access Point: punkt dostępu klienta, czyli nazwa i adres IP roli klastra, pod którymi użytkownicy łączą się z usługą niezależnie od tego, który węzeł ją prowadzi
CQL	Cassandra Query Language: język zapytań Cassandry, podobny do SQL
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
SAP	Systeme, Anwendungen und Produkte in der Datenverarbeitung: niemiecki producent oprogramowania klasy ERP, właściciel silnika HANA i przejętego wcześniej Sybase ASE
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Po co to istnieje

Wybór silnika bazodanowego bywa traktowany jako wybór między „SQL" a „NoSQL", jakby to były dwie równoważne opcje. W praktyce „SQL" oznacza tu jeden konkretny model danych — relacyjny — a „NoSQL" jest workiem, do którego wrzucono wszystko, co tym modelem nie jest, choć te „wszystko inne" różni się między sobą bardziej, niż różnią się od modelu relacyjnego.

Rodziny alternatywne wobec modelu relacyjnego nie są nowością ostatnich lat — wyrosły w drugiej połowie lat dwutysięcznych z konkretnego problemu: firmy w rodzaju Google i Amazon obsługiwały ruch, przy którym pojedynczy serwer relacyjny przestawał wystarczać, a jednoczesne wymaganie ścisłej spójności i pracy na tysiącach maszyn naraz okazało się trudne do pogodzenia. Publikacje takie jak artykuł Google o systemie Bigtable (2006) czy Amazona o systemie Dynamo (2007) opisały architektury zaprojektowane pod kątem skalowania poziomego kosztem części gwarancji, które model relacyjny traktował jako oczywiste — i to na tych architekturach wprost wzorują się dzisiejsze Cassandra i Redis.

> **Pojęcie: model danych.** Sposób, w jaki system organizuje dane koncepcyjnie i w jaki się je odpytuje — nie fizyczny format pliku na dysku, tylko kształt, w którym programista myśli o danych: jako o tabelach powiązanych kluczami, jako o samodzielnych dokumentach, jako o parach klucz-wartość. Wybór modelu danych determinuje, jakie pytania da się zadać łatwo, a jakie — trudno albo wcale.

Rodzina relacyjna

Opisana w szczegółach na stronie Model relacyjny: tabela, wiersz, kolumna, dziedzina — dane w tabelach powiązanych kluczami, odpytywane językiem SQL, z silnym naciskiem na spójność i integralność referencyjną wymuszaną przez sam silnik. Przedstawiciele na tej półce: PostgreSQL, Oracle Database, Microsoft SQL Server, MySQL, Sybase (SAP ASE). Sensowna, gdy dane mają jasną, powtarzalną strukturę i gdy zależy nam na tym, żeby silnik sam pilnował spójności między powiązanymi tabelami — księgowość, zamówienia, dane kadrowe, wszystko, gdzie „ten wiersz nie może istnieć bez tamtego" jest regułą biznesową, nie sugestią.

Rodzina dokumentowa

> **Pojęcie: dokument.** Samoopisujący się rekord, zwykle w formacie JSON albo jego binarnej odmianie (BSON w MongoDB), który trzyma wszystkie dane dotyczące jednej encji w jednej strukturze, łącznie z zagnieżdżonymi listami i obiektami — bez rozbijania ich na osobne tabele połączone kluczami obcymi. Przedstawiciel: MongoDB. Sensowna, gdy struktura danych jest zmienna między rekordami (nie każdy produkt w katalogu ma te same atrybuty) albo gdy dana encja i tak jest zawsze czytana i zapisywana w całości — profil użytkownika z całą jego konfiguracją w jednym dokumencie, zamiast złączenia pięciu tabel za każdym odczytem.

Rodzina kolumnowa (szeroko-kolumnowa)

> **Pojęcie: magazyn szeroko-kolumnowy.** Model, w którym dane grupowane są wierszami zidentyfikowanymi kluczem, ale w przeciwieństwie do tabeli relacyjnej każdy wiersz może mieć inny zestaw kolumn, a dane fizycznie są przechowywane i kompresowane kolumna po kolumnie, nie wiersz po wierszu — co czyni odczyt jednej kolumny z milionów wierszy bardzo tanim, kosztem odczytu całego wiersza naraz. Przedstawiciel: Cassandra. Sensowna przy bardzo dużym wolumenie zapisów rozłożonych na wiele serwerów naraz, gdzie odporność na awarię pojedynczego węzła i skalowanie poziome liczą się bardziej niż złożone zapytania łączące wiele encji — telemetria z urządzeń, logi zdarzeń, dane czasowe o wysokiej częstotliwości zapisu.

Rodzina klucz-wartość

> **Pojęcie: magazyn klucz-wartość.** Najprostszy z modeli — baza to w istocie ogromna mapa, gdzie każdemu kluczowi odpowiada jedna wartość, odczytywana i zapisywana wyłącznie po tym kluczu, bez języka zapytań w stylu SQL. Przedstawiciel: Redis. Sensowna jako warstwa bardzo szybkiego dostępu do danych o prostej strukturze dostępu — cache wyników zapytań, sesje użytkowników, liczniki, kolejki — nie jako główne, trwałe źródło prawdy dla danych o złożonych powiązaniach.

Czwarta, mniej liczna na tej półce rodzina to bazy grafowe (np. Neo4j), gdzie węzły i krawędzie między nimi są danymi pierwszej klasy, a nie czymś symulowanym kluczami obcymi — sensowne tam, gdzie pytanie brzmi „jak te rzeczy są ze sobą powiązane" (sieci społecznościowe, wykrywanie oszustw, rekomendacje), a nie „ile mamy tych rzeczy". W modelu relacyjnym odpowiedź na pytanie „znajomi znajomych znajomych" wymaga kilku złączeń tej samej tabeli z samą sobą i szybko robi się kosztowna; w bazie grafowej to przejście po krawędziach, które silnik wykonuje jako swoją podstawową operację.

Kiedy która ma sens

Rodzina	Mocna strona	Cena

Relacyjna	integralność wymuszana przez silnik, elastyczne zapytania ad hoc (JOIN, agregacje)	schemat trzeba zaprojektować z góry, skalowanie poziome jest trudniejsze
Dokumentowa	elastyczna struktura, wygodna gdy encja jest czytana/zapisywana w całości	złączenia między dokumentami są niewygodne lub niemożliwe, spójność między dokumentami zwykle po stronie aplikacji
Kolumnowa	bardzo wysoka przepustowość zapisu, skalowanie poziome na wielu węzłach	ograniczony język zapytań, złączenia praktycznie nie istnieją
Klucz-wartość	ekstremalna szybkość dla prostego dostępu po kluczu	brak zapytań po czymkolwiek innym niż klucz, brak relacji między wartościami

Jak to działa: schemat wymuszony a schemat zakładany

> **Pojęcie: schema-on-write.** Podejście, w którym silnik wymusza zgodność ze strukturą (typy kolumn, ograniczenia) w chwili zapisu — `INSERT` niezgodny ze schematem tabeli jest odrzucany, zanim dane trafią na dysk. Tak działa każdy silnik relacyjny na tej półce.

> **Pojęcie: schema-on-read.** Podejście, w którym silnik przy zapisie nie narzuca żadnej konkretnej struktury — przyjmie dowolny dokument JSON — a to, czy struktura jest zgodna z oczekiwaniami, ujawnia się dopiero w momencie, gdy aplikacja próbuje odczytać konkretne pole. Tak działa MongoDB: dwa dokumenty w tej samej kolekcji mogą mieć zupełnie inny zestaw pól, i silnik nie ma nic przeciwko.

Różnica ta nie jest kwestią „lepiej" czy „gorzej" — to przesunięcie w czasie momentu, w którym niezgodność strukturalna staje się widoczna. Schema-on-write wychwytuje błąd danych natychmiast, kosztem sztywności. Schema-on-read pozwala na ewolucję struktury bez migracji całej kolekcji naraz, kosztem tego, że błąd (brakujące pole, zły typ) ujawnia się dopiero w kodzie aplikacji, czasem długo po zapisaniu danych.

Twierdzenie CAP w jednym akapicie

> **Pojęcie: twierdzenie CAP.** Dla systemu rozproszonego, w którym dane są replikowane na kilku węzłach, twierdzenie CAP mówi, że w chwili, gdy sieć między węzłami się przerywa (partycja), system musi wybrać między dalszym odpowiadaniem na zapytania z potencjalnie nieaktualnymi danymi (dostępność) a odmową odpowiedzi, dopóki nie ma pewności co do spójności danych (spójność) — nie może mieć jednocześnie pełni obu. To nie jest wybór, którego dokonuje się raz na zawsze przy projektowaniu systemu — dobre systemy rozproszone pozwalają dostroić to zachowanie per operację, a odporność na partycję sieci nie jest w praktyce opcją do odrzucenia, tylko warunkiem brzegowym każdej architektury rozproszonej, więc realny wybór dotyczy tego, co system robi w chwili partycji, a nie tego, którą z trzech liter „wybiera raz na stałe" — wbrew popularnemu uproszczeniu „pick two of three".

Dlaczego „NoSQL" to zła nazwa

Nazwa definiuje kategorię przez negację — mówi, czym coś nie jest, a nie czym jest. MongoDB, Cassandra i Redis różnią się między sobą modelem danych, sposobem skalowania i typowym zastosowaniem bardziej, niż niektóre z nich różnią się od baz relacyjnych w konkretnych aspektach — Cassandra ma własny język zapytań (CQL) składniowo bardzo zbliżony do SQL, a MongoDB od pewnego czasu oferuje interfejs SQL-podobny do swojej agregacji. Trzymanie się nazwy „NoSQL" sugeruje istnienie jednej alternatywy dla SQL, podczas gdy w rzeczywistości istnieje kilka niezależnych modeli danych, z których każdy trzeba oceniać osobno pod kątem konkretnego problemu — nie da się wybrać „NoSQL" tak, jak wybiera się jeden konkretny silnik.

Częste błędy

Wybór modelu dokumentowego z powodu mody, nie potrzeby. Dane, które mają stałą, powtarzalną strukturę i wymagają częstych złączeń między encjami (zamówienia i ich pozycje, użytkownicy i ich role), w modelu dokumentowym kończą albo jako zagnieżdżone, zduplikowane dane w wielu dokumentach naraz, albo jako ręcznie odtworzone złączenia po stronie aplikacji — czyli dokładnie to, co model relacyjny robi za darmo.

Traktowanie „schema-on-read" jako braku schematu w ogóle. Baza dokumentowa nie wymusza jednej struktury na poziomie silnika, ale to nie znaczy, że struktury nie ma — aplikacja, która czyta dokumenty, wciąż zakłada konkretne pola i typy. Różnica jest w momencie, kiedy niezgodność struktury ujawnia się jako błąd: przy zapisie (relacyjna) czy dopiero przy odczycie, w kodzie aplikacji (dokumentowa).

Odczytanie twierdzenia CAP jako „trzeba wybrać dwie z trzech cech na zawsze". To popularne uproszczenie ignoruje, że tolerancja na partycję sieci nie jest opcjonalna w systemie

rozproszonym — pytanie brzmi wyłącznie, co system robi w chwili awarii sieci, nie które dwie litery z trzech ktoś sobie wybrał przy projektowaniu.

□□ Ćwiczenia

> Poniższe zadania nie wymagają zainstalowanego żadnego konkretnego silnika — chodzi o rozpoznanie modelu, nie o składnię.

Pytania kontrolne

1. System rejestruje dziesiątki tysięcy odczytów z czujników IoT na sekundę, a typowe zapytanie to „pokaż wszystkie odczyty czujnika X z ostatniej godziny". Która rodzina pasuje najlepiej i dlaczego?

Odpowiedź

Rodzina kolumnowa (np. Cassandra) — bardzo wysoka przepustowość zapisu rozłożona na wiele węzłów, a typowe zapytanie jest proste (jeden klucz identyfikujący czujnik, zakres czasu), więc nie potrzeba bogatego języka złączeń, za to potrzeba skalowania poziomego, którego model relacyjny na pojedynczym serwerze nie daje równie łatwo.

2. Dlaczego nazwanie Cassandry i MongoDB „tym samym typem bazy" jest błędem, mimo że obie często pojawiają się pod wspólną etykietą „NoSQL"?

Odpowiedź

Bo reprezentują dwa różne modele danych — Cassandra jest magazynem szeroko-kolumnowym zoptymalizowanym pod masowy, rozproszony zapis i proste zapytania po kluczu, a MongoDB jest bazą dokumentową zoptymalizowaną pod przechowywanie i odczyt złożonych, samodzielnych rekordów. Łączy je wyłącznie to, czym nie są — modelem relacyjnym — co jest dokładnie tym problemem z nazwą „NoSQL", o którym mówi ta strona.

Powiązane: Model relacyjny: tabela, wiersz, kolumna, dziedzina · Replikacja: synchroniczna, asynchroniczna, logiczna, fizyczna

Źródła: E. Brewer, prezentacja o twierdzeniu CAP (PODC 2000) · dokumentacja MongoDB, rozdz. „Data Modeling" · dokumentacja Apache Cassandra, rozdz. „Data Modeling"