INSERT: jeden wiersz, wiele wierszy, INSERT ... SELECT

`INSERT` dokłada do tabeli nowe wiersze — jeden, wiele naraz albo skopiowane z innej tabeli zapytaniem — a ta strona pokazuje wszystkie trzy warianty razem z tym, jak odczytać wartość, którą silnik nadał automatycznie, i jak wstawić dane masowo bez czekania godzinami.

◫ Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
DML	Data Manipulation Language: polecenia SQL zmieniające dane (INSERT, UPDATE, DELETE)
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
OID	Object Identifier: numer, którym w SNMP wskazujemy jedną wartość urządzenia, na przykład stan baterii UPS
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

◫ ZMIENIA SYSTEM

Wstawienie jednego wiersza z jawną listą kolumn

```
INSERT INTO klienci (nazwa, miasto, nip) VALUES ('Elektro-System Kowalczyk', 'Poznań',
'7123456780');
```

Wstawiamy jeden nowy wiersz do tabeli `klienci`, podając wprost, które kolumny wypełniamy i w jakiej kolejności. Kolumny, których nie wymieniliśmy — tu `id` — silnik wypełnia sam, jeśli

mają zdefiniowaną wartość domyślną albo generator, opisany osobno na stronie Sekwencje, IDENTITY, AUTO_INCREMENT.

Przykład:

```
INSERT 0 1
```

`psql` po udanym `INSERT` wypisuje dwie liczby: pierwsza to identyfikator obiektu OID, dziś w praktyce zawsze zero, bo tabele nie używają już OID domyślnie; druga to liczba wstawionych wierszy.

OID = Object Identifier: numer, którym w SNMP wskazujemy jedną wartość urządzenia, na przykład stan baterii UPS

ZMIENIA SYSTEM

Wstawienie bez jawnej listy kolumn — i dlaczego tego unikamy

```
INSERT INTO klienci VALUES (7, 'Nowy Klient Sp. z o.o.', 'Łódź', '7291234567');
```

Ten sam efekt bez wymieniania nazw kolumn — silnik przypisuje wartości po kolei, w takiej kolejności, w jakiej kolumny zostały zdefiniowane w `CREATE TABLE`. Zapytanie zadziała dzisiaj, ale przestanie działać poprawnie, gdy tylko ktoś doda nową kolumnę w środku definicji tabeli albo zmieni jej kolejność — wtedy wartości podstawią się pod złe kolumny bez żadnego ostrzeżenia ze strony silnika, dokładnie tak jak opisuje to strona Model relacyjny: tabela, wiersz, kolumna, dziedzina przy okazji atrybutów. Dlatego w tej książce `INSERT` zawsze pojawia się z jawną listą kolumn, nawet gdy wypełniamy wszystkie naraz.

Przykład:

```
INSERT 0 1
```

ZMIENIA SYSTEM

Wstawienie wielu wierszy jednym poleceniem

```
INSERT INTO produkty (nazwa, kategoria, cena) VALUES
    ('Podkładka pod mysz', 'akcesoria', 35.00),
    ('Głośniki komputerowe', 'elektronika', 180.00),
    ('Fotel gamingowy', 'meble', 1450.00);
```

Jedna instrukcja `INSERT` z listą krotek po `VALUES`, oddzielonych przecinkami, wstawia wiele wierszy naraz. Silnik traktuje to jako jedną operację — jedno sprawdzenie ograniczeń na starcie transakcji niejawnej, jeden zapis do dziennika transakcyjnego zamiast trzech osobnych — co ma znaczenie praktyczne opisane niżej w sekcji o ładowaniu masowym.

Przykład:

```
INSERT 0 3
```

◫◫ ZMIENIA SYSTEM

Kopiowanie danych między tabelami: INSERT ... SELECT

```
INSERT INTO produkty_archiwalne (nazwa, kategoria, cena)
SELECT nazwa, kategoria, cena FROM produkty WHERE kategoria = 'meble';
```

`INSERT ... SELECT` zastępuje listę `VALUES` zapytaniem `SELECT` — silnik wstawia do tabeli docelowej dokładnie te wiersze, które zwróciłoby to zapytanie, uruchomione osobno. Liczba i typy kolumn po obu stronach muszą się zgadzać pozycyjnie, nie muszą zgadzać się nazwami. To najczęstszy sposób przenoszenia danych między tabelą roboczą a archiwalną, między środowiskiem testowym a raportowym, albo przy scalaniu dwóch baz po migracji.

Przykład:

```
INSERT 0 3
```

Trzy produkty z kategorii `meble` — `Biurko regulowane`, `Fotel biurowy` i `Lampka biurkowa` — trafiają do `produkty_archiwalne` w jednym poleceniu, bez odczytywania ich najpierw do aplikacji i wstawiania z powrotem.

◫◫ ZMIENIA SYSTEM

Odczytanie nadanego identyfikatora: RETURNING (PostgreSQL)

```
INSERT INTO klienci (nazwa, miasto, nip) VALUES ('Global Logistics Sp. z o.o.',
'Szczecin', '3830123456') RETURNING id;
```

`RETURNING` każe silnikowi zwrócić fragment wiersza, który właśnie wstawił — najczęściej klucz główny nadany automatycznie, ale równie dobrze dowolną inną kolumnę albo wyrażenie. Bez tego aplikacja musiałaby wykonać drugie zapytanie, żeby dowiedzieć się, jaki `id` dostał nowy klient — `RETURNING` załatwia to w tym samym poleceniu, bez dodatkowego zapytania i

bez ryzyka, że między dwoma poleceniami wstawi się jeszcze czyjś inny wiersz.

Przykład:

```
 id
----
  7
(1 row)
```

◫◫ ZMIENIA SYSTEM

Odczytanie nadanego identyfikatora: OUTPUT (SQL Server)

```
INSERT INTO klienci (nazwa, miasto, nip)
OUTPUT INSERTED.id
VALUES (N'Global Logistics Sp. z o.o.', N'Szczecin', '3830123456');
```

Odpowiednik `RETURNING` w SQL Server — pseudo-tabela `INSERTED` udostępnia wiersze, które polecenie właśnie wstawiło, `OUTPUT` decyduje, które ich kolumny trafiają do wyniku zwróconego klientowi. Litera `N` przed literałem tekstowym każe SQL Server potraktować go jako `NVARCHAR`, nie `VARCHAR` — istotne dla polskich znaków w kolumnach unikodowych.

Przykład:

```
id
-----------
7


(1 rows affected)
```

◫◫ **SQL** = Structured Query Language: język zapytań do relacyjnych baz danych

◫◫ TYLKO ODCZYT

Ładowanie masowe z pliku: COPY (PostgreSQL)

```
COPY klienci (nazwa, miasto, nip) FROM '/sciezka/do/pliku/klienci.csv' WITH (FORMAT csv,
HEADER true);
```

`COPY` czyta plik bezpośrednio po stronie serwera, z pominięciem parsowania pojedynczych poleceń `INSERT` i większości narzutu transakcyjnego przypadającego na wiersz. To narzędzie do ładowania tysięcy albo milionów wierszy naraz, nie do pojedynczych wstawień w logice aplikacji.

Przykład:

```
COPY 1500
```

◫◫ ZMIENIA SYSTEM

Ładowanie masowe z pliku: BULK INSERT (SQL Server)

```
BULK INSERT NazwaTabeli
FROM 'C:\SciezkaDoPliku\dane.csv'
WITH (FORMAT = 'CSV', FIRSTROW = 2, FIELDTERMINATOR = ',');
```

Odpowiednik `COPY` w SQL Server, uruchamiany z poziomu samego silnika — plik musi być widoczny dla konta usługi bazy, nie tylko dla klienta, który wydaje polecenie. `FIRSTROW = 2` pomija wiersz nagłówkowy pliku CSV.

Przykład:

```
(1500 rows affected)
```

Narzędzie `bcp` robi to samo z poziomu wiersza poleceń systemu operacyjnego, bez łączenia się przez `sqlcmd`, i nadaje się lepiej do automatyzacji w skryptach zewnętrznych względem bazy.

◫◫ **SQL** = Structured Query Language: język zapytań do relacyjnych baz danych · **CSV** = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami

◫◫ TYLKO ODCZYT

Ładowanie masowe z pliku: LOAD DATA INFILE (MySQL)

```
LOAD DATA INFILE '/sciezka/do/pliku/klienci.csv'
INTO TABLE klienci
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
```

```
IGNORE 1 ROWS
(nazwa, miasto, nip);
```

`IGNORE 1 ROWS` pomija nagłówek pliku, dokładnie jak `FIRSTROW = 2` w `BULK INSERT`. Plik domyślnie musi leżeć na serwerze, do którego MySQL ma dostęp; wariant `LOAD DATA LOCAL INFILE` czyta plik z maszyny klienta kosztem dodatkowego ustawienia po stronie serwera i sterownika.

Przykład:

```
Query OK, 1500 rows affected (0.42 sec)
Records: 1500  Deleted: 0  Skipped: 0  Warnings: 0
```

TYLKO ODCZYT

Ładowanie masowe z pliku: SQL*Loader (Oracle)

```
sqlldr userid=nazwa_usera/hasło@NazwaSID control=klienci.ctl log=klienci.log
```

SQL*Loader to osobne narzędzie wiersza poleceń, nie polecenie SQL wewnątrz `sqlplus` — czyta plik kontrolny `.ctl`, w którym opisujemy format danych wejściowych i mapowanie kolumn, a wynik i ewentualne odrzucone wiersze zapisuje do plików `.log` i `.bad` obok pliku danych.

Przykład:

```
SQL*Loader: Release 19.0.0.0.0 - Production

Path used:      Conventional
Commit point reached - logical record count 1500

Table KLIENCI:
  1500 Rows successfully loaded.
```

SQL = Structured Query Language: język zapytań do relacyjnych baz danych

TYLKO ODCZYT

Cztery narzędzia ładowania masowego

Silnik	Narzędzie	Uruchamiane z
PostgreSQL	`COPY`	wnętrza sesji SQL (`psql \copy` — po stronie klienta)
SQL Server	`BULK INSERT` / `bcp`	`BULK INSERT` wewnątrz T-SQL, `bcp` z wiersza poleceń OS
MySQL / MariaDB	`LOAD DATA INFILE`	wnętrza sesji SQL
Oracle	SQL*Loader (`sqlldr`)	osobny program wiersza poleceń OS

Uważaj

- `COPY` w czystej postaci działa po stronie serwera i wymaga, żeby plik leżał tam, gdzie serwer go widzi — `\copy` w `psql` to osobny wariant klienta, który czyta plik lokalnie i przesyła dane przez połączenie.
- SQL*Loader jest jedynym z czterech narzędzi, które nie jest poleceniem SQL — to zewnętrzny program, uruchamiany z powłoki systemu operacyjnego, z własnym formatem pliku sterującego.

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 · **OS** = Operating System: system operacyjny

Po co to istnieje

Pojęcie: wstawienie wiersza. Dodanie nowej krotki do instancji relacji, czyli do aktualnego zbioru danych trzymanego przez tabelę, bez zmiany schematu tej relacji — schemat, czyli definicję kolumn i ich typów, zmienia dopiero `ALTER TABLE`, opisany na stronie ALTER TABLE: co blokuje tabelę, a co nie. `INSERT` jest jedynym poleceniem DML, które tworzy wiersze; `UPDATE` i `DELETE` operują wyłącznie na wierszach, które już istnieją.

Wybór między pojedynczym `INSERT`, wieloma wierszami w jednym poleceniu a `INSERT ... SELECT` zależy od tego, skąd pochodzą dane. Pojedynczy wiersz pasuje do formularza aplikacji, w którym użytkownik dodaje jeden rekord. Wiele wierszy w jednym poleceniu pasuje do niewielkiego zestawu danych znanego już w chwili pisania zapytania. `INSERT ... SELECT` pasuje wszędzie tam, gdzie dane już istnieją w bazie, tylko w innej tabeli albo w innym kształcie.

Jak to działa

> **Pojęcie: autocommit.** Tryb pracy połączenia, w którym każde pojedyncze polecenie SQL dostaje własną, niejawną transakcję, zatwierdzaną automatycznie zaraz po jego wykonaniu — bez jawnego `BEGIN` i `COMMIT` ze strony aplikacji. To domyślny tryb w większości sterowników i narzędzi klienckich, wygodny dla pojedynczych zapytań, ale kosztowny przy wielu operacjach z rzędu, bo każda z nich osobno czeka na zapis potwierdzenia do dziennika transakcyjnego, opisanego dokładniej na stronie Log transakcyjny: WAL, redo, transaction log.

Dziesięć tysięcy osobnych poleceń `INSERT` wykonanych w autocommit to dziesięć tysięcy osobnych zatwierdzeń — każde z nich wymaga fizycznego zapisu do dziennika transakcyjnego na dysku, zanim silnik potwierdzi sukces klientowi, i przy typowym opóźnieniu pojedynczego zatwierdzenia rzędu kilku do kilkudziesięciu milisekund suma tych opóźnień potrafi rozciągnąć się na godzinę, nawet jeśli same dane są niewielkie. Ta sama liczba wierszy wstawiona w jednej transakcji — jawne `BEGIN`, seria `INSERT`-ów, jeden `COMMIT` na końcu — wymaga tylko jednego zatwierdzenia na końcu całej partii, więc te same dziesięć tysięcy wierszy trafiają do tabeli w czasie liczonym w sekundach, nie w godzinach. Mechanizm transakcji, `BEGIN` i `COMMIT`, ma własną, pełną stronę: BEGIN, COMMIT, ROLLBACK, SAVEPOINT.

`INSERT ... SELECT` i ładowanie masowe (`COPY`, `BULK INSERT`, `LOAD DATA INFILE`, SQL*Loader) omijają ten problem inaczej — nie tyle grupując zatwierdzenia, ile w ogóle pomijając narzut pojedynczego polecenia SQL na wiersz: parsowanie tekstu zapytania, sprawdzanie planu wykonania i komunikację sieciową między klientem a serwerem, powtórzone tysiąc razy zamiast wykonane raz dla całej partii.

Częste błędy

Naruszenie ograniczenia unikalności przy wstawianiu istniejącego NIP-u.

```
ERROR:  duplicate key value violates unique constraint "klienci_nip_key"
DETAIL:  Key (nip)=(5261234567) already exists.
```

Wiersz z takim samym NIP-em już istnieje w tabeli — ograniczenie `UNIQUE` na kolumnie `nip` odrzuca drugi. Naprawa to sprawdzenie, czy klient już istnieje (`SELECT` po `nip` przed `INSERT`em), albo użycie jednej z konstrukcji opisanych na stronie MERGE / UPSERT — jak to robi każdy silnik, które łączą sprawdzenie i wstawienie w jedną operację odporną na współbieżność.

Niezgodna liczba kolumn między listą a wartościami.

```
ERROR:  INSERT has more expressions than target columns
LINE 1: INSERT INTO klienci (nazwa, miasto) VALUES ('Test', 'Radom...
                                                    ^
```

Lista `VALUES` ma więcej elementów niż lista kolumn w nawiasie po nazwie tabeli — częsty efekt kopiowania fragmentu z innego polecenia bez dopasowania obu list.

Naruszenie klucza obcego przy wstawianiu zamówienia dla nieistniejącego klienta.

```
ERROR:  insert or update on table "zamowienia" violates foreign key constraint
"fk_zamowienia_klient"
DETAIL:  Key (klient_id)=(99) is not present in table "klienci".
```

`klient_id` równy `99` nie istnieje w tabeli `klienci` — kolejność wstawiania musi iść od tabeli nadrzędnej do podrzędnej, dokładnie jak opisuje to strona Klucze: główny, obcy, unikalny, naturalny, sztuczny.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — jawna lista kolumn po zmianie schematu

Do tabeli `produkty (id, nazwa, kategoria, cena)` dodajemy nową kolumnę `dostawca_id INTEGER` na końcu listy definicji. Które z dwóch poleceń nadal wstawi dane poprawnie: `INSERT INTO produkty VALUES (7, 'Kamera internetowa', 'elektronika', 249.00)` czy `INSERT INTO produkty (nazwa, kategoria, cena) VALUES ('Kamera internetowa', 'elektronika', 249.00)`?

Odpowiedź

Żadne z nich nie zadziała bez zmian — pierwsze poda zbyt mało wartości względem nowej, pięciokolumnowej definicji tabeli i skończy się błędem `INSERT has more target columns than expressions` albo podobnym, w zależności od silnika; drugie zadziała poprawnie, bo jawna lista kolumn w ogóle nie wspomina o `dostawca_id`, więc silnik wstawi tam wartość domyślną albo `NULL`. To jest właśnie powód, dla którego jawna lista kolumn jest bezpieczniejsza — nie psuje

się przy dodaniu kolumny, tylko przy usunięciu którejś z wymienionych.

Pytania kontrolne

1. Czym różni się `INSERT` z listą `VALUES` od `INSERT ... SELECT`?

Odpowiedź

`VALUES` podaje dane wprost w treści polecenia, znane już w chwili pisania zapytania. `INSERT ... SELECT` pobiera dane z wyniku innego zapytania, uruchamianego przez silnik w tym samym poleceniu — źródłem może być inna tabela, złączenie kilku tabel albo ta sama tabela z innym filtrem.

2. Dlaczego `RETURNING` i `OUTPUT` oszczędzają jedno zapytanie w porównaniu z osobnym odczytem po `INSERT`zie?

Odpowiedź

Bez nich aplikacja musiałaby po `INSERT`cie wykonać drugie zapytanie, żeby poznać wartość nadaną automatycznie, na przykład `SELECT currval(...)` albo odczyt po jakimś innym warunku — a między tymi dwoma poleceniami inny wiersz mógłby zostać wstawiony równolegle, co przy niewłaściwym warunku odczytu groziłoby pobraniem cudzego identyfikatora. `RETURNING`/`OUTPUT` zwraca dokładnie ten wiersz, który dane polecenie właśnie wstawiło, w tej samej operacji.

Powiązane: UPDATE: i dlaczego zawsze piszemy WHERE zanim napiszemy SET · Sekwencje, IDENTITY, AUTO_INCREMENT · MERGE / UPSERT — jak to robi każdy silnik · BEGIN, COMMIT, ROLLBACK, SAVEPOINT

Źródła: dokumentacja PostgreSQL, rozdział „INSERT” i „COPY” · dokumentacja Microsoft, „BULK INSERT (Transact-SQL)” · dokumentacja MySQL, „LOAD DATA Statement” · dokumentacja Oracle, „SQL*Loader Concepts” · ISO/IEC 9075-2:2016 § 14.11

Revision #47
Created 2026-09-24 22:42:11 UTC by Tomasz Konieczny
Updated 2026-10-26 14:11:14 UTC by Tomasz Konieczny