Cheatography

SQL Cheat Sheet

by sigeud via cheatography.com/188214/cs/39451/

SQL DML (Data Manipulation Language)

SELECT	`SELECT * FROM tabela;`
INSERT	`INSERT INTO tabela (kolumna1, kolumna2) VALUES ('wartość1', 'wartość2');`
UPDATE	`UPDATE tabela SET kolumna1 = 'nowa_ wartość' WHERE warunek;`
DELETE	`DELETE FROM tabela WHERE warunek;`

DML umożliwia manipulację danymi w bazie danych, obejmuje operacje takie jak wstawianie, modyfikowanie, usuwanie i pobieranie danych z bazy.

SQL DQL (Data Query Language)

SELECT	`SELECT kolumna1, kolumna2 FROM tabela WHERE warunek;`
DISTINCT	`SELECT DISTINCT kolumna FROM tabela;`
ORDER BY	`SELECT kolumna1, kolumna2 FROM tabela ORDER BY kolumna1 ASC;`
GROUP BY	`SELECT kolumna, COUNT(*) FROM tabela GROUP BY kolumna;`

DQL służy do pobierania danych z bazy danych, umożliwia formułowanie zapytań, które pozwalają na selekcję i wyszukiwanie określonych danych z tabel. SQL DQL oferuje również inne funkcje, takie jak filtrowanie danych za pomocą warunków, łączenie tabel, operatory logiczne i wiele innych. DQL umożliwia elastyczne i precyzyjne pobieranie danych z bazy danych.

SQL DDL (Data Definition Language)

CREATE (TABLE, DATABASE, ...)	`CREATE TABLE tabela ( kolumna1 typ_da nych, kolumna2 typ_da nych, ... );`
CREATE VIEW	`CREATE VIEW widok AS SELECT kolumna1, kolumna2 FROM tabela WHERE warunek;`
ALTER TABLE (ADD, DROP, ALTER)	`ALTER TABLE tabela ADD kolumna typ_da nych;`
DROP (TABLE, DATABASE, ...)	`DROP TABLE tabela;`

DDL służy do definiowania struktury i organizacji bazy danych. DDL umożliwia tworzenie, modyfikację i usuwanie obiektów bazy danych, takich jak tabele, widoki, indeksy, sekwencje itp. SQL DDL oferuje również inne instrukcje, takie jak tworzenie indeksów, sekwencji, ograniczeń integralności, procedur składowanych i wiele innych. DDL umożliwia definiowanie struktury i organizacji bazy danych, co jest kluczowe dla tworzenia i zarządzania danymi w systemach baz danych.

By **sigeud**
cheatography.com/sigeud/

Published 4th July, 2023.
Last updated 9th July, 2023.

SQL DCL (Data Control Language)

GRANT	używana do udzielania uprawnień użytko-wnikom w bazie danych	`GRANT SELECT, INSERT ON tabela TO użytkownik ;`
REVOKE	używana do cofania uprawnień użytko-wnikom w bazie danych	`REVOKE DELETE ON tabela FROM użytko wnik;`
GRANT ROLE	używana do nadawania roli użytko-wnikowi w bazie danych	`GRANT rola TO użytko wnik;`
COMMIT	używana do zatwie-rdzania transakcji, czyli trwałego zapisania zmian wprowa-dzonych w bazie danych	`COMMIT;`

DCL odpowiada za kontrolę dostępu do bazy danych, obejmuje instrukcje umożliwiające zarządzanie uprawnieniami użytkowników, zabezpieczeniami i transakcjami w bazie danych. SQL DCL jest ważnym elementem zarządzania bazą danych, umożliwiając kontrolę dostępu, zabezpieczenia i trwałość danych. DCL zapewnia mechanizmy niezbędne do zarządzania uprawnieniami użytko-wników oraz zabezpieczeń w celu ochrony danych w bazie.

SQL TCL (Transactional Control Language)

COMMIT	używana do zatwierdzania bieżącej transa-kcji, co oznacza trwałe zapisanie wprowadzonych zmian w bazie danych	`COMMIT;`
ROLLBACK	jest używana do wycofywania bieżącej transa-kcji, czyli cofania wprowadzonych zmian do stanu poprzedniego	`ROLLBACK;`
SAVEPOINT	używana do tworzenia punktu zapisu w trakcie trwania transakcji	`SAVEPOINT punkt_ zapisu ;`

SQL TCL (Transactional Control Language) (cont)

RELEASE SAVEPOINT	używana do usunięcia punktu zapisu utworzonego za pomocą instrukcji SAVEPOINT	`RELEASE SAVEPOINT punkt_ zapisu;`

TCL obejmuje instrukcje do zarządzania transakcjami w bazie danych. TCL umożliwia rozpoczynanie, zatwierdzanie i wycofywanie transakcji oraz zarządzanie punktami zapisu. SQL TCL zapewnia kontrolę nad transakcjami w bazie danych, umożliwiając rozpoczynanie, zatwierdzanie i wycofywanie zmian. Dzięki instrukcjom TCL można zapewnić spójność i integralność danych oraz zarządzać punktami zapisu w celu odtworzenia stanu transakcji w przypadku potrzeby.

Widoki (Views)

Widoki (ang. views) to wirtualne tabele utworzone na podstawie wyników zapytań SQL. Są to logiczne reprezentacje danych, które można wykorzystać do uproszczenia złożonych zapytań, ukrycia szczegółów implementacyjnych i zapewnienia dostępu do odpowiednich danych dla różnych użytkowników.
Widoki są tworzone na podstawie istniejących tabel lub innych widoków, a ich struktura jest definiowana przez zapytanie SELECT. Po utworzeniu widoku można odwoływać się do niego tak, jakby był to zwykła tabela w bazie danych.

```
CREATE VIEW SalesR eport AS
SELECT ProductID, SUM(Qu antity * Price) AS TotalSale
s
FROM Sales
WHERE OrderDate BETWEEN '2022- 01-01' AND '2022- 12-3
1'
```

Widoki (Views) (cont)

```
GROUP BY ProductID;
SELECT * FROM SalesR eport;
```

Łączenie zbiorów (tabel)

UNION	łączy wyniki dwóch zapytań SELECT i zwraca unikalne wiersze. Kolumny w obu zapytaniach muszą być zgodne pod względem liczby i typów danych	`SELECT column1, column2 FROM table1 UNION SELECT column1, column2 FROM table2;`
UNION ALL	wykonuje łączenie wyników zapytań bez usuwania duplikatów	`SELECT column1, column2 FROM table1 WHERE condition UNION ALL SELECT column1, column2 FROM table2 WHERE condition;`

Łączenie zbiorów (tabel) (cont)

INTERSECT	zwraca tylko te wiersze, które występują jednocześnie w wynikach obu zapytań SELECT	`SELECT column1, column2 FROM table1 INTERSECT SELECT column1, column2 FROM table2;`
EXCEPT (lub MINUS)	zwraca tylko te wiersze, które występują w wyniku pierwszego zapytania SELECT, ale nie występują w wyniku drugiego zapytania SELECT	`SELECT column1, column2 FROM table1 EXCEPT SELECT column1, column2 FROM table2;`

Łączenie tabel

JOIN (INNER JOIN)	`SELECT Orders.Or derID, Custom ers.Cu sto merName FROM Orders INNER JOIN Customers ON Orders.Cu sto merID = Custom ers.Cu sto m erID;`

Łączenie tabel (cont)

LEFT JOIN	`SELECT Custom ers.Cu sto mer Name, Orders.Or derID FROM Customers LEFT JOIN Orders ON Custom ers.Cu sto merID = Orders.Cu sto m erID;`
RIGHT JOIN	`SELECT Custom ers.Cu sto mer Name, Orders.Or derID FROM Customers RIGHT JOIN Orders ON Custom ers.Cu sto merID = Orders.Cu sto m erID;`
FULL JOIN	`SELECT Custom ers.Cu sto mer Name, Orders.Or derID FROM Customers FULL JOIN Orders ON Custom ers.Cu sto merID = Orders.Cu sto m erID;`

Łączenie tabel (cont)

Zagnieżdżone zapytania (subqueries)	`SELECT * FROM Table1 WHERE Column1 IN ( SELECT Column1 FROM Table2 WHERE Column2 = 'wartość' );`
Warunki złączenia (JOIN conditions) w klauzuli WHERE	`SELECT * FROM Table1, Table2 WHERE Table1.Co lumn1 = Table2.Co lumn1;`
Funkcje skalarne	`SELECT Table1.Co lumn1, Table1.Co lumn2, ( SELECT Column3 FROM Table2 WHERE Table2.Co lumn1 = Table1.Co lumn1) AS Column3 FROM Table1;`
CROSS JOIN	`SELECT * FROM Table1 CROSS JOIN Table2;`

Łączenie tabel (cont)

NATURAL JOIN (automatycznie łączy dwie tabele na podstawie wspólnych nazw kolumn)	`SELECT * FROM Table1 NATURAL JOIN Table2;`

Funkcje okna (window functions)

Funkcje okna (window functions) w SQL umożliwiają przeprowadzenie
na grupach wierszy, zwanych "oknami", wewnątrz wynikowego zestawu
Pozwalają na wykonywanie zaawansowanych operacji agregacyjnych,
gowych i analitycznych w kontekście określonych grup wierszy. Oto kilk
kluczowych informacji na temat funkcji okna wraz z przykładami:

1. Składnia:

Funkcje okna są zwykle wywoływane za pomocą specjalnej składni, któ
zawiera nawiasy okrągłe wokół funkcji oraz klauzulę OVER, która określ
jakiej podstawie powinna być obliczana funkcja okna.

Przykład:

```
SELECT order_id, custom er_id, order_ total, SUM(or
tal)
OVER (PARTITION BY custom er_id) AS sum_or der _tota
FROM orders;
```

Rezultat:

order_id | customer_id | order_total | sum_order_total

---

1 | 101 | 50.00 | 150.00
2 | 101 | 75.00 | 150.00
3 | 102 | 100.00 | 250.00
4 | 103 | 200.00 | 200.00

Funkcje okna (window functions) (cont)

5 | 103 | 50.00 | 200.00

2. PARTITION BY:

Klauzula PARTITION BY w funkcji okna służy do podziału wynikowego zestawu danych na grupy, na podstawie których funkcje okna są obliczane. Każda grupa otrzymuje niezależne obliczenia funkcji okna.

Przykład:

```
SELECT column1, column2, AVG(co lumn3)
OVER (PARTITION BY column1, column2) AS avg_co lumn
3
FROM table;
```

W tym przykładzie AVG() jest funkcją okna, która oblicza średnią wartości column3 dla każdej kombinacji unikalnych wartości column1 i column2.

3. ORDER BY:

Klauzula ORDER BY w funkcji okna służy do określenia porządku sortowania w obrębie każdej grupy wierszy. Określa to, na podstawie których kolumn mają być obliczane funkcje okna.

Przykład:

```
SELECT column1, column2, RANK()
OVER (PARTITION BY column1
ORDER BY column2 DESC) AS rank_c olumn2
FROM table;
```

4. Funkcje okna vs group by

Funkcje okna (window functions) (cont)

Funkcje okna są bardziej elastycznym narzędziem, które pozwala na wykonywanie obliczeń na grupach wierszy w obrębie wynikowego zestawu danych, zachowując jednocześnie pełną strukturę wynikową. Klauzula GROUP BY jest bardziej odpowiednia, gdy chcesz dokonać agregacji danych i otrzymać zredukowany wynik dla każdej grupy.

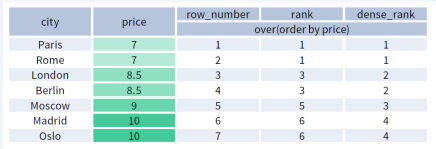
Bardziej zaawansowane funkcje agregujące

RANK() / DENSE_-RANK() / ROW_NUMBER()	Przypisuje wartość rankingu, gęstego rankingu lub numeru wiersza do każdego wiersza w wynikach zapytania	`SELECT kolumna, RANK() OVER (ORDER BY kolumna DESC) AS Rank FROM tabela;`
NTILE()	Dzieli zestaw danych na równą liczbę grup i przypisuje numer grupy do każdego wiersza	`SELECT kolumna, NTILE(4) OVER (ORDER BY kolumna) AS GroupN umb FROM tabela;`
CUME_DIST()	Oblicza kumula-cyjną wartość dystry-bucji dla danego wiersza w zestawie danych	`SELECT kolumna, CUME_D IST() OVER (ORDER BY kolumna) AS Cumula tiv ion FROM tabela;`

Bardziej zaawansowane funkcje agregujące (cont)

FIRST_-VALUE() / LAST_VALUE()	Zwraca pierwszą lub ostatnią wartość z określonej kolumny w ramach grupy	`SELECT kolumna, FIRST_ VAL UE( kol umna) OVER (PARTITION BY inna_k olumna ORDER BY kolejn a_k olumna) AS FirstValue FROM tabela;`
LAG() / LEAD()	Zwraca wartość z poprze-dniego lub następnego wiersza w zestawie danych	`SELECT kolumna, LAG(ko lumna) OVER (ORDER BY kolejn a_k olumna) AS Previo usVal ue FROM tabela;`
NTH_VALUE()	Zwraca wartość z określ-onego wiersza w określonym oknie, gdzie numer wiersza jest podany	`NTH_VA LUE (co lumn, n) OVER (PARTITION BY ...)`

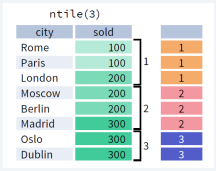
row_number() / rank() / dense_rank()



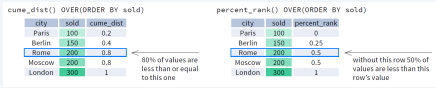
city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

https://learnsql.com/blog/sql-window-functions-cheat-sheet/

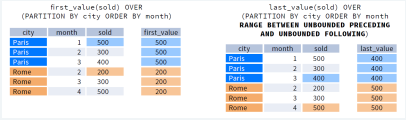
ntile(n)



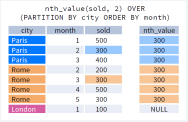
percent_rank() / cume_dist()



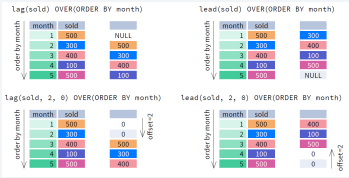
first_value(expr) / last_value(expr)



nth_value(expr, n)



lead() / lag()



lead(expr, offset, default) - the value for the row offset rows after the current; offset and default are optional; default values: offset = 1, default = NULL

lag(expr, offset, default) - the value for the row offset rows before the current; offset and default are optional; default values: offset = 1, default = NULL

Funkcje WITHIN GROUP

Wyrażenie "WITHIN GROUP" jest często używane w kontekście funkcji agregujących do określania porządku sortowania w wynikach agregacji. Pozwala na kontrolę kolejności sortowania wartości w grupie. Within Group jest szczególnie przydatne, gdy chcemy kontrolować kolejność sortowania wartości wewnątrz grupy w wyniku agregacji i dostosować je do naszych wymagań.

- STRING_AGG, która łączy wartości w jednym łańcuchu znaków, zdefiniowanym separatorem.

Przykład:

```
SELECT custom er_id, STRING _AG G(o rde r_id, ', ' ORDER BY order_id) WITHIN G
ROUP (ORDER BY order_id) AS order_list
FROM orders
GROUP BY custom er_id;
```

Wynik:

```
+-----------------+--------------+
| customer_id | order_list |
| 1001 | 1, 2 |
| 1002 | 3, 4, 5 |
+-----------------+-----------------+
```

Funkcje WITHIN GROUP (cont)

- PERCENTILE_CONT():

```
SELECT job_title, PERCEN TIL E_C ONT (0.5) WITHIN GR(
ry)
AS median _salary
FROM employees
GROUP BY job_title;
```

wynik:

```
+--------------+-----------------+
| job_title | median_salary |
+--------------+-----------------+
| Manager | 5500.00 |
| Developer | 4000.00 |
| Analyst | 3500.00 |
+--------------+-----------------+
```

-PERCENTILE_DISC() - służy do obliczania wartości procentylowej dys
grupy danych. Wartość procentylowa dyskretna to wartość danych, któ
onemu percentylowi.

```
SELECT PERCEN TIL E_D ISC (0.75)
WITHIN GROUP (ORDER BY score)
AS percen til e_value
```

Funkcje WITHIN GROUP (cont)

```
FROM scores;
+----------------+
| percentile_value |
+----------------+
| 85 |
+----------------+
select
percen til e_d isc (ar ray [0.1, 0.5, 0.9])
within group (order by liczba _pa saz erow) as
kwantyle from wnioski
```

- PERCENTILE_DISC() vs PERCENTILE_CONT()

Funkcja PERCENTILE_DISC() zwraca dyskretną wartość danych, która odpowiada określonemu percentylowi. Oznacza to, że wartość procentylowa jest jedną z istniejących wartości w zbiorze danych. Na przykład, jeśli mamy zbiór liczb [1, 2, 3, 4, 5], to wartość procentylowa dyskretna dla 75% będzie wynosić 4, ponieważ 4 jest jedną z wartości w zbiorze.

Funkcja PERCENTILE_CONT() zwraca wartość procentylową ciągłą, która jest interpolacją między wartościami w zbiorze danych. Oznacza to, że wartość procentylowa może być wartością, która nie występuje w zbiorze danych. Na przykład, dla zbioru liczb [1, 2, 3, 4, 5], wartość procentylowa ciągła dla 75% może wynosić 4.75, co oznacza interpolację między 4 a 5 na podstawie proporcji.

- MODE() - służy do znalezienia trybu (wartości, która występuje najczęściej) w zestawie danych. Oto przykład użycia funkcji MODE() wraz z wynikami

```
SELECT category, MODE() WITHIN GROUP (ORDER BY category)
AS mode_c ategory
FROM products
GROUP BY category;
+----------+----------------+
| category | mode_category |
+----------+----------------+
| Electronics | Electronics |
| Clothing | Clothing |
| Books | Books |
+----------+----------------+
```

Statystyczne funkcje agregujące

CORR(dependent, indepedent)	Oblicza współczynnik korelacji między dwiema zmiennymi	`SELECT CORR(s ales, advert ising) FROM sales_ data;`
COVAR_POP() / COVAR_SAMP()	Oblicza kowariancję między dwiema zmiennymi	`SELECT COVAR_ POP (in come, expen nce FROM financ ial _data;`

Statystyczne funkcje agregujące (cont)

STDDEV-_POP() / STDDEV-_SAMP() / STDDEV()	Oblicza odchylenie standa-rdowe danych	`SELECT STDDEV _PO P(t emp era turn) AS popula tio n_stddev, STDDEV _SA MP( tem per ature) AS sample _st ddev, STDDEV (te mpe rature) AS stddev FROM weathe r_data;`
VAR_POP() / VAR_SAMP() / VARIANCE()	Oblicza wariancję danych	`SELECT VAR_PO P(s ales) AS population_variance, VAR_SA MP( sales) AS sample_variance, VARIAN CE( sales) AS variance FROM sales_ data;`
REGR_INTE-RCEPT()	Oblicza wartość punktu przecięcia linii regresji liniowej	`SELECT REGR_I NTE RCE PT( sales, advert ising) AS inter cept FROM sales_ data;`

Statystyczne funkcje agregujące (cont)

REGR_S LOPE()	Oblicza współc-zynnik nachylenia linii regresji liniowej.	`SELECT REGR_S LOP E(s ales, adver ope FROM sales_ data;`

Operatory

- Operator IN: Operator IN służy do porównywania wartości z jednym lub więcej wyników podzapytania. Zwraca prawdę (true), jeśli wartość znajduje się w wynikach podzapytania, w przeciwnym razie zwraca fałsz (false).

Przykład:

```
SELECT column1
FROM table1
WHERE column2 IN (SELECT column3 FROM table2);
```

- Operator NOT IN, który wykonuje negację operatora IN. Operator NOT IN zwraca prawdę (true), jeśli wartość nie znajduje się w wynikach podzapytania.

Przykład:

```
SELECT column1
FROM table1
WHERE column2 NOT IN (1, 2, 3);
```

- Operator = / <> (NOT EQUAL): Operator = służy do porównywania równości między dwiema wartościami. Operator <> (lub !=) sprawdza, czy wartości są różne od siebie.

Przykład:

```
SELECT column1
FROM table1
```

Operatory (cont)

```
WHERE column2 = 'value';
SELECT column1
FROM table1
WHERE column2 <> 'value';
```

- Operator EXISTS: Operator EXISTS sprawdza, czy podzapytanie zwraca jakiekolwiek wiersze. Jeśli podzapytanie zwraca wyniki, to operator EXISTS zwraca prawdę (true), w przeciwnym razie zwraca fałsz (false). Operator EXISTS jest często stosowany do sprawdzania istnienia rekordów w innej tabeli.

Przykład:

```
SELECT column1
FROM table1
WHERE EXISTS (SELECT column2 FROM table2 WHERE column2 = table1.co lumn3)
;
```

- Operator ANY / ALL: Operator ANY porównuje wartość z wynikami podzapytania przy użyciu określonego operatora porównania (np. >, <, =). Operator ANY zwraca prawdę (true), jeśli porównanie jest prawdziwe dla co najmniej jednego wyniku podzapytania. Operator ALL zwraca prawdę (true), jeśli porównanie jest prawdziwe dla wszystkich wyników podzapytania.

Przykład:

```
SELECT column1
FROM table1
WHERE column2 > ANY (SELECT column3 FROM table2);
SELECT column1
FROM table1
WHERE column2 = ALL (SELECT column3 FROM table2);
```

Operatory (cont)

- Operator BETWEEN: Operator BETWEEN służy do porównywania wartości w określonym zakresie. Sprawdza, czy wartość znajduje się pomiędzy dwiema innymi wartościami. Może być używany w klauzuli WHERE.

Przykład:

```
SELECT column1
FROM table1
WHERE column2 BETWEEN 10 AND 20;
```

- Operator LIKE: Operator LIKE jest używany do porównywania wartości z wyrażeniem regularnym lub wzorcem. Często stosuje się go w klauzuli WHERE w celu wyszukiwania wzorców w tekście.

Przykład:

```
SELECT column1
FROM table1
WHERE column2 LIKE 'A%';
```

- Operator ILIKE jest używany w niektórych systemach baz danych (np. PostgreSQL) i działa podobnie jak operator LIKE, ale jest nieczuły na wielkość liter. Operator ILIKE porównuje wartości z wyrażeniem regularnym lub wzorcem, ignorując różnice w wielkości liter.

Przykład:

```
SELECT column1
FROM table1
WHERE column2 ILIKE 'a%';
```

- Operator IS NULL / IS NOT NULL: Operator IS NULL służy do sprawdzania, czy wartość w kolumnie jest pusta (NULL). Operator IS NOT NULL sprawdza, czy wartość nie jest pusta.

Przykład:

Operatory (cont)

```
SELECT column1
FROM table1
WHERE column2 IS NULL;
```

Transakcje SQL

Transakcje SQL to sekwencje operacji, które muszą być wykonane jako całość - albo wszystkie zostaną zatwierdzone i zapisane w bazie danych, albo żadna z nich zostanie zapisana. Transakcje są używane do zapewnienia spójności danych i utrzymania integralności bazy danych.

W SQL transakcje są zazwyczaj używane w kontekście operacji na bazie danych, takich jak wprowadzanie zmian, aktualizacja danych, czytanie i zapisywanie danych. Oto podstawowe pojęcia związane z transakcjami SQL:

1. BEGIN/START TRANSACTION: Rozpoczyna nową transakcję.

```
BEGIN TRANSA CTION;
```

2. COMMIT: Zatwierdza transakcję i wprowadza wszystkie jej zmiany do bazy danych.

```
COMMIT;
```

3. ROLLBACK: Anuluje transakcję i cofa wszystkie jej zmiany, przywracając bazę danych do stanu przed rozpoczęciem transakcji.

```
ROLLBACK;
```

4. SAVEPOINT: Utworzenie punktu kontrolnego w trakcie transakcji, który można użyć do późniejszego przywrócenia transakcji do określonego stanu.

```
SAVEPOINT savepo int _name;
```

5. RELEASE SAVEPOINT: Usuwa określony punkt kontrolny utworzony w trakcie transakcji.

```
RELEASE SAVEPOINT savepo int _name;
```

Transakcje SQL (cont)

6. ROLLBACK TO SAVEPOINT: Cofa transakcję do określonego punktu kontrolnego, usuwając wszystkie zmiany po tym punkcie.

```
ROLLBACK TO SAVEPOINT savepo int _name;
```

Transakcje SQL zapewniają spójność danych w przypadku błędów, awarii systemu lub konfliktów jednoczesnego dostępu do danych. Pozwalają również na grupowanie operacji w większe jednostki logiczne i wprowadzanie zmian do bazy danych w kontrolowany sposób.

Warto zauważyć, że nie wszystkie systemy bazodanowe obsługują transakcje w taki sam sposób, a składnia i zachowanie mogą się różnić. Należy skonsultować się z dokumentacją systemu bazodanowego, którego używasz, aby uzyskać szczegółowe informacje na temat transakcji SQL w danym systemie.

Najprostszą transkacją w SQL Server jest pojedyncza instrukcja modyfikacji danych.

```
UPDATE Person.Ad dress
SET Addres sLine1 = 'Prosta 51'
WHERE AddressID = 1
```

Transakcje z wieloma operacjami (Explicit Transactions)

```
BEGIN TRANSA CTION
UPDATE Person.Ad dress
SET Addres sLine1 = 'Prosta 51'
WHERE AddressID = 1
UPDATE Person.Ad dress
SET Addres sLine1= 'Przyo kopowa 31'
```

WHERE AddressID = 2`

COMMIT TRANSACTION`

https://www.plukasiewicz.net/Artykuly/SQL_Transactions

Procedury, funkcje i triggery

Procedury, funkcje i triggery są elementami języka SQL, które pozwalają na definiowanie i wykonywanie niestandardowych operacji i logiki w bazie danych. Oto ich krótkie opisy:

1. Procedury:

- Procedury to zbiorcze instrukcje SQL, które są zdefiniowane i przechowywane w bazie danych.
- Procedury mogą przyjmować parametry wejściowe, wykonywać operacje na danych, a następnie zwracać wyniki lub modyfikować dane w bazie.
- Procedury mogą być wywoływane przez inne zapytania SQL lub programy aplikacyjne.
- Przykład:

```
CREATE PROCEDURE GetCus tom erO rders
    @custo merId INT
AS
BEGIN
    SELECT * FROM Orders WHERE CustomerId = @custo merId;
END;
```

2. Funkcje:

- Funkcje są podobne do procedur, ale różnią się sposobem użycia i zwracanymi wartościami.
- Funkcje zwracają wartość, która może być wykorzystana w zapytaniach SQL, wyrażeniach lub instrukcjach.
- Mogą być wykorzystywane jako część zapytań SELECT, warunków WHERE, wyrażeń CASE itp.
- Funkcje mogą być skalarnymi funkcjami (zwracające pojedynczą wartość) lub funkcjami tabelarycznymi (zwracające tabelę wynikową).
- Przykład:

```
CREATE FUNCTION GetOrd erTotal
    (@orderId INT)
RETURNS DECIMA L(10, 2)
AS
BEGIN
    DECLARE @total DECIMA L(10, 2);
    SELECT @total = SUM(Qu antity * Price) FROM Orde
    RETURN @total;
END;
```

3. Triggery:

- Triggery są automatycznymi reakcjami na zdarzenia lub operacje w ba
izacja lub usuwanie danych.
- Trigger to blok kodu SQL, który jest wywoływany w odpowiedzi na okr
- Triggery mogą być definiowane dla tabel i uruchamiane przed lub po w
- Mogą być wykorzystywane do wprowadzania logiki biznesowej, spraw
itp.
- Przykład:

```
CREATE TRIGGER Update Ord erTotal
ON OrderItems
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    UPDATE Orders
```

Procedury, funkcje i triggery (cont)

```
    SET TotalA mount = (SELECT SUM(Qu antity * Price) FROM inserted WHERE Or derId =
insert ed.O rd erId)
    FROM Orders INNER JOIN inserted ON Orders.Or derId = inserted.Or derId;
END;
```

Procedury, funkcje i triggery są potężnymi narzędziami w SQL, które pozwalają na tworzenie bardziej zaawansowanych operacji, logiki biznesowej i automatyzacji w bazach danych. Ich zastosowanie zależy od konkretnych wymagań i scenariuszy w projekcie bazodanowym.

* "@" przed nazwą oznacza, że jest to nazwa parametru wejściowego funkcji.

sprawdzanie typów danych

- Aby sprawdzić typy danych w tabeli, można skorzystać z polecenia DESCRIBE lub DESC. Oto przykład:

```
DESCRIBE tabela;
```

Lub

```
DESC tabela;
```

To polecenie wyświetli informacje o strukturze tabeli, w tym nazwy kolumn, typy danych, rozmiary kolumn itp.

- Alternatywnie, można również skorzystać z zapytania SHOW COLUMNS FROM tabela, które również zwróci informacje o typach danych w tabeli. Oto przykład:

```
SHOW COLUMNS FROM tabela;
```

Oba te polecenia zwrócą wyniki, które przedstawiają typy danych dla każdej kolumny w tabeli, wraz z innymi informacjami takimi jak nazwy kolumn, klucze główne, atrybuty null itp.

- Aby sprawdzić typ danych na konkretnej kolumnie w tabeli, można użyć polecenia DESCRIBE z podaniem nazwy kolumny. Oto przykład:

```
DESCRIBE tabela nazwa_ kol umny;
```

sprawdzanie typów danych (cont)

Na przykład, jeśli chcemy sprawdzić typ danych kolumny "imie" w tabeli "uzytkownicy", użyjemy polecenia:

```
DESCRIBE uzytkownicy imie;
```

Polecenie to zwróci informacje dotyczące typu danych dla określonej kolumny, takie jak typ danych, rozmiar, atrybuty null itp.

SELECT a SELECT "

Różnica między SELECT *FROM tabela a SELECT '* FROM tabela polega na tym, jak zostaną zwrócone dane.

- SELECT * FROM tabela: Wykonanie tego zapytania zwróci wszystkie kolumny (wszystkie dane) z tabeli. Każda kolumna zostanie zwrócona jako oddzielna kolumna wynikowa. Przykład:

```
CREATE TABLE osoba (
  id INT,
  imie VARCHA R(50),
  nazwisko VARCHA R(50)
);
INSERT INTO osoba (id, imie, nazwisko)
VALUES (1, 'John', 'Doe'),
       (2, 'Jane', 'Smith');
SELECT * FROM osoba;
```

Wynik:

id	imie	nazwisko
1	John	Doe

SELECT a SELECT '' (cont)

| 2 | Jane | Smith |

- SELECT '*' FROM tabela: Wykonanie tego zapytania zwróci pojedynczą kolumnę, w której każdy wiersz będzie zawierał tekst '' (gwiazdkę). Oznacza to, że nie zostaną zwrócone rzeczywiste dane z tabeli, tylko powtórzony znak '', tyle razy, ile jest wierszy w tabeli.

Przykład:

```
CREATE TABLE osoba (
  id INT,
  imie VARCHA R(50),
  nazwisko VARCHA R(50)
);
INSERT INTO osoba (id, imie, nazwisko)
VALUES (1, 'John', 'Doe'),
       (2, 'Jane', 'Smith');
SELECT '*' FROM osoba;
```

Wynik:

'*'
'*'
'*'

Jak widać, w drugim przypadku zwracane są tylko powtórzone gwiazdki, nie uwzględniając rzeczywistych danych z tabeli. Należy pamiętać, że SELECT '*' FROM tabela może mieć zastosowanie w niektórych specjalnych przypadkach, np. jako szybka metoda sprawdzenia, czy tabela zawiera dane, ale nie jest to typowe użycie przy wybieraniu rzeczywistych danych z tabeli.

operator LIKE

W operatorze `LIKE` w języku SQL można używać różnych wzorców. Oto kilka najczęściej stosowanych wzorców wraz z ich opisem:

1. `%` (znak procentu):

Symbol `%` odpowiada dowolnej liczbie znaków (również zero znaków). Może być używany na początku, na końcu lub w środku wzorca. Przykłady:

- `'%abc'` - pasuje do ciągów, które kończą się na 'abc'
- `'abc%'` - pasuje do ciągów, które zaczynają się od 'abc'
- `'%abc%'` - pasuje do ciągów, które zawierają 'abc' gdziekolwiek

2. `_` (podkreślnik):

Symbol `_` odpowiada dokładnie jednemu znakowi. Może być używany na początku, na końcu lub w środku wzorca. Przykłady:

- `'a_'` - pasuje do dwuznakowych ciągów, które zaczynają się od 'a'
- `'_bc'` - pasuje do dwuznakowych ciągów, które kończą się na 'bc'
- `'_b_'` - pasuje do trzyznakowych ciągów, które mają 'b' na drugiej pozycji

3. `[]` (klasy znaków):

Klasy znaków pozwalają na określenie zbioru dopuszczalnych znaków w danej pozycji. Przykłady:

- `'a[bc]d'` - pasuje do ciągów, które mają 'a', a następnie 'b' lub 'c', a potem 'd'
- `'[0-9]abc'` - pasuje do ciągów, które mają cyfrę od 0 do 9, a następnie 'abc'

4. `[^]` (negacja klasy znaków):

Negacja klasy znaków oznacza dopasowanie znaku, który nie jest w podanym zbiorze. Przykład:

- `'a[^bc]d'` - pasuje do ciągów, które mają 'a', a następnie znak, który nie jest 'b' ani 'c', a potem 'd'

operator LIKE (cont)

Te wzorce można również łączyć i zagnieżdżać, aby tworzyć bardziej zaawansowane wyrażenia dopasowania w operatorze `LIKE`. Warto pamiętać, że składnia i dostępność wzorców mogą się nieco różnić w zależności od używanej bazy danych.

W zwykłym operatorze LIKE w języku SQL nie ma wbudowanej składni [a-z], która reprezentuje zakres liter od "a" do "z" (czyli wszystkie małe litery w alfabecie angielskim). Operator LIKE obsługuje jedynie proste wzorce z wykorzystaniem symboli % i _, oraz klasy znaków [].

Jednak w niektórych bazach danych, takich jak PostgreSQL i MySQL, można użyć wyrażeń regularnych wraz z operatorem REGEXP lub RLIKE do bardziej zaawansowanych dopasowań, w tym zakresów liter.

Przykład z użyciem wyrażeń regularnych w PostgreSQL:

```
SELECT column _name
FROM table_name
WHERE column _name ~ '^[a-z]$';
```

Przykład z użyciem wyrażeń regularnych w MySQL:

```
SELECT column _name
FROM table_name
WHERE column _name REGEXP '^[a-z]$';
```

W powyższych przykładach wyrażenie regularne ^[a-z]$ oznacza dopasowanie jednej małej litery od "a" do "z" w kolumnie. Uwaga, że składnia i obsługa wyrażeń regularnych mogą się różnić w zależności od konkretnej bazy danych.

Wzorce takie jak % i _ wciąż mają takie same znaczenie w przypadku operatora ILIKE. Jednak operatory klasy znaków [ ] i [^] nie będą działać w dokładnie ten sam sposób jak w przypadku operatora LIKE. Zachowanie tych wzorców w kontekście ILIKE zależy od konkretnej bazy danych.

is null = '', not null, <> ''

W przypadku pustych wartości (NULL) w języku SQL, operator `IS NULL` jest używany do sprawdzania, czy wartość w kolumnie jest pusta (NULL), a operator `IS NOT NULL` jest używany do sprawdzania, czy wartość w kolumnie nie jest pusta (nie jest NULL).

Natomiast porównanie pustej wartości z pustym ciągiem znaków `" "` (empty string) jest czymś innym. Pusty ciąg znaków `" "` jest traktowany jako niepusty (non-NULL) ciąg, ale nie ma żadnej zawartości. Oznacza to, że porównanie `=` z pustym ciągiem znaków nie jest równoważne z operatorem `IS NULL`.

Przykład:

```
SELECT column _name
FROM table_name
WHERE column _name = '';
```

W powyższym przykładzie, `column _name = ''` porównuje wartość w kolumnie `column _name` z pustym ciągiem znaków `''`. To sprawdzi, czy wartość jest pustym ciągiem, ale nie sprawdzi, czy jest NULL.

Podobnie, porównanie `<>` z pustym ciągiem znaków nie jest równoważne z operatorem `IS NOT NULL`. Porównanie `<>` z pustym ciągiem znaków sprawdzi, czy wartość w kolumnie jest różna od pustego ciągu znaków, ale nadal może zawierać wartość NULL.

Jeśli chcesz sprawdzić, czy wartość w kolumnie jest pusta lub NULL, powinieneś nadal używać operatorów `IS NULL` lub `IS NOT NULL`.

Wniosek: Porównanie pustego ciągu znaków `''` nie jest równoważne z operatorem `IS NULL`, a porównanie `<> ''` nie jest równoważne z operatorem `IS NOT NULL`. Dlatego zaleca się używanie operatorów `IS NULL` i `IS NOT NULL` do sprawdzania pustych wartości w SQL.

Funkcje agregujące z groupby a w funkcji okna

Użycie funkcji agregujących z klauzulą `GROUP BY` i w funkcjach okna ma kilka istotnych różnic:

1. Działanie na grupach: Użycie funkcji agregujących z klauzulą `GROUP BY` pozwala na grupowanie danych na podstawie określonych kolumn i obliczanie agregatów dla każdej grupy. Wyniki są podzielone na grupy, a funkcje agregujące są obliczane dla każdej grupy osobno.

Przykład:

```
SELECT depart ment, SUM(sa lary) AS total_ salary
FROM employees
GROUP BY depart ment;
```

Użycie funkcji okna, z drugiej strony, nie grupuje danych w taki sposób. Funkcje okna obliczają wartości dla każdego wiersza w wyniku, niezależnie od grupowania. Funkcje okna działają na całym zbiorze wynikowym i nie powodują podziału na grupy.

Przykład:

```
SELECT employ ee_id, last_name, salary, SUM(sa lary
)
OVER (PARTITION BY depart ment) AS total_ salary
FROM employees;
```

W powyższym przykładzie, funkcja okna `SUM` jest używana do obliczania sumy wynagrodzenia dla każdego pracownika w ramach danego departamentu, ale nie powoduje to podziału na grupy.

2. Zakres obliczeń: Użycie funkcji agregujących z klauzulą `GROUP BY` ogranicza wyniki do grup, dla których obliczone są agregaty. Każda grupa jest reprezentowana przez jeden wiersz wynikowy.

W przypadku funkcji okna, wynikowe wiersze odpowiadają wszystkim wierszom z wyniku zapytania, a funkcje okna obliczają wartości na podstawie określonych okien (partycji) i kolejności sortowania. Wynik zwracany przez funkcje okna zachowuje wszystkie wiersze, a nie tworzy jednego wiersza na grupę.

Funkcje agregujące z groupby a w funkcji okna (cont)

3. Składnia zapytania: Użycie funkcji agregujących z klauzulą `GROUP BY` wymaga wyraźnego określenia kolumn, które mają być uwzględnione w grupowaniu. Natomiast użycie funkcji okna odbywa się za pomocą odpowiedniej składni w ramach klauzuli `SELECT`, bez konieczności podawania klauzuli `GROUP BY`.

Przykład:

```
SELECT depart ment, AVG(sa lary) AS avg_salary
FROM employees
GROUP BY depart ment;
SELECT employ ee_id, last_name, salary, AVG(sa lary
)
OVER (PARTITION BY depart ment) AS avg_salary
FROM employees;
```

W przypadku funkcji okna, nie ma potrzeby podawania kolumny `depa rtment` w klauzuli `SELECT`. Funkcja okna jest wywoływana bezpośrednio w zapytaniu `SELECT`, a wynik jest automatycznie obliczany dla każdego wiersza.

Podsumowując, użycie funkcji agregujących z klauzulą `GROUP BY` jest skoncentrowane na grupowaniu danych i obliczaniu agregatów dla poszczególnych grup, podczas gdy funkcje okna działają na poziomie pojedynczego wiersza i pozwalają na obliczanie wartości w ramach zdefiniowanych okien i kolejności sortowania bez podziału na grupy. Ostateczny wybór między tymi podejściami zależy od wymagań zapytania i oczekiwanych wyników.

Tabela DUAL

Tabela "DUAL" jest specjalną tabelą w niektórych bazach danych, takich jak Oracle, która jest często używana do wykonywania zapytań testowych, testowania wyrażeń SQL lub pobierania wartości bez odwoływania się do rzeczywistych tabel w bazie danych.
Tabela "DUAL" zawiera tylko jedną kolumnę o nazwie "DUMMY" i jedno zawsze jedno wiersz o wartości "X" lub "Y". Ta tabela jest zawsze dostępna w bazie danych i nie wymaga jej tworzenia.
Przykład użycia tabeli "DUAL":

```
SELECT 'Hello, World!' AS message FROM DUAL;
```

W tym przykładzie wykonujemy prosty zapytanie, które zwraca wartość 'Hello, World!' jako kolumnę "message". Ponieważ nie potrzebujemy się odwoływać do żadnej konkretnej tabeli, używamy tabeli "DUAL" jako źródła danych. Zapytanie to zwróci tylko jeden wiersz z jedną kolumną.
Tabela "DUAL" jest szczególnie przydatna w przypadkach, gdy chcemy wykonać prosty testowy zapytanie, przetestować składnię SQL lub wygenerować wartości stałe bez konieczności odwoływania się do istniejących tabel w bazie danych.
Warto jednak zaznaczyć, że nie wszystkie bazy danych obsługują tabelę "DUAL" jako domyślną tabelę specjalną. Niektóre bazy danych, takie jak MySQL czy SQL Server, nie mają wbudowanej tabeli "DUAL". W takich przypadkach można użyć innego podejścia, na przykład używając instrukcji `SELECT` bez odwoływania się do jakiejkolwiek tabeli, lub tworząc tymczasową tabelę do testowania zapytań.

Pseudo kolumny

Pseudo-kolumny to specjalne kolumny dostępne w języku SQL, które zawierają dodatkowe informacje na temat danych lub kontekstu zapytania. Pseudo-kolumny nie są fizycznymi kolumnami w tabeli, ale są dostępne do użycia w zapytaniach SQL. Poniżej przedstawiam kilka popularnych pseudo-kolumn:
1. `ROWNUM` (Oracle) lub `ROW_NU MBER()` (inny dialekt SQL):
Ta pseudo-kolumna zawiera numer porządkowy wiersza w zbiorze wynikowym. Jest szczególnie przydatna przy ograniczaniu wyników zapytania do określonej liczby wierszy lub wykonywaniu paginacji.
Przykład:

```
SELECT ROWNUM, first_ name, last_name
FROM employees
WHERE ROWNUM <= 10;
```

2. `ROWID` (Oracle) lub `CTID` (PostgreSQL):
Ta pseudo-kolumna zawiera unikalny identyfikator wiersza w tabeli. Jest używana głównie do bezpośredniego odwoływania się do konkretnych wierszy w celu aktualizacji lub usunięcia.
Przykład:

```
UPDATE employees
SET salary = 5000
WHERE ROWID = 'AAABB BCCC';
```

3. `SYSDATE` (Oracle) lub `CURREN T_T IME STAMP` (inny dialekt SQL):
Ta pseudo-kolumna zawiera aktualną datę i czas na serwerze bazy danych.
Przykład:

```
SELECT order_id, order_ date, SYSDATE AS curren t_da
te
FROM orders;
```

Pseudo kolumny (cont)

4. `LEVEL` (Oracle) lub `GENERATION` (inny dialekt SQL):
Ta pseudo-kolumna jest używana w zapytaniach rekurencyjnych lub do określenia poziomu hierarchii w zapytaniach związanych z drzewem.
Przykład:

```
SELECT employ ee_id, last_name, LEVEL
FROM employees
START WITH employ ee_id = 1
CONNECT BY PRIOR employ ee_id = manage r_id;
```

5. `OBJECT_ID` (Oracle): Pseudo-kolumna `OBJECT_ID` jest używana w Oracle Database do zwracania identyfikatora obiektu bazy danych. Może być używana w zapytaniach dotyczących metadanych lub dostępu do informacji o konkretnym obiekcie bazy danych, takim jak tabela, widok, procedura, itp.
Przykład:

```
SELECT object _name, object _type
FROM all_ob jects
WHERE object_id = 12345;
```

6. `OBJECT _VALUE` (Oracle): Pseudo-kolumna `OBJECT _VALUE` jest używana w Oracle XML DB do zwracania wartości XML dla obiektu XML w bazie danych. Może być stosowana w zapytaniach, które operują na danych XML przechowywanych w bazie danych Oracle.
Przykład:

```
SELECT object_id, object _value
FROM xml_table;
```

Pseudo kolumny (cont)

7. `ORA_ROWSCN` (Oracle): Pseudo-kolumna `ORA_ROWSCN` jest używana w Oracle Database do zwracania System Change Number (SCN) dla wiersza. SCN jest unikalnym identyfikatorem przypisanym do każdego wiersza w bazie danych i może być używany do monitorowania zmian i śledzenia historii danych.
Przykład:

```
SELECT rowid, ora_rowscn
FROM my_table;
```

8. `XMLDATA` (Oracle): Pseudo-kolumna `XMLDATA` jest używana w Oracle XML DB do zwracania danych XML w formacie XMLType. Może być stosowana w zapytaniach, które operują na danych XML przechowywanych w bazie danych Oracle.
Przykład:

```
SELECT xmldata
FROM xml_table;
```

9. `CURRVAL` i `NEXTVAL` (Oracle): Pseudo-kolumny `CURRVAL` i `NEXTVAL` są używane w Oracle Database w połączeniu z sekwencjami (sequences) do pobierania bieżącej lub następnej wartości sekwencji. `CURRVAL` zwraca bieżącą wartość sekwencji, podczas gdy `NEXTVAL` zwraca następną wartość sekwencji.
Przykład:

```
SELECT my_seq uen ce.N EXTVAL
FROM dual;
```

10. `CONNEC T_B Y_I SCYCLE` (Oracle): Pseudo-kolumna `CONNEC T_B Y_I SCYCLE` jest używana w Oracle Database w kontekście zapytań związanych z drzewami (`CONNECT BY`) do oznaczenia, czy wystąpił cykl w hierarchii. Zwraca wartość 1, jeśli wierzchołek jest częścią cyklu, lub 0 w przeciwnym razie.

Pseudo kolumny (cont)

Przykład:

```
SELECT employ ee_id, last_name, CONNEC T_B Y_I SCYC
LE
FROM employees
START WITH employ ee_id = 1
CONNECT BY PRIOR employ ee_id = manage r_id;
```

Pseudo-kolumny różnią się w zależności od dialektu SQL i używanej bazy danych. Nie wszystkie pseudo-kolumny są dostępne we wszystkich bazach danych, dlatego zawsze warto sprawdzić dokumentację swojej konkretnej bazy danych, aby dowiedzieć się, jakie pseudo-kolumny są dostępne i jak ich używać.

Self Join

Self join w języku SQL jest techniką polegającą na łączeniu tabeli z samą sobą. Oznacza to, że w zapytaniu używamy tej samej tabeli jako dwóch odrębnych instancji, które łączymy ze sobą na podstawie warunków zdefiniowanych w klauzuli ON.

Przykładem może być tabela "employees" zawierająca informacje o pracownikach, gdzie mamy kolumny "employee_id", "first_name", "last_name" i "manager_id", w której "manager_id" wskazuje na identyfikator przełożonego danego pracownika.

Przykład zapytania self join, które zwraca informacje o pracownikach i ich przełożonych, może wyglądać tak:

```
SELECT e.firs t_name AS employ ee_ fir st_ name, e.last _name AS employ ee_ la
s t_name,
       m.firs t_name AS manage r_f irs t_name, m.last _name AS manage r_l ast
_name
FROM employees e
JOIN employees m ON e.mana ger_id = m.empl oye e_id;
```

Self Join (cont)

W powyższym przykładzie tabela "employees" jest łączona z samą sobą na podstawie warunku e.manager_id = m.employee_id. Otrzymujemy wynik, w którym dla każdego pracownika zostaje wyświetlone imię i nazwisko pracownika oraz imię i nazwisko jego przełożonego.

Self join jest przydatny w sytuacjach, gdy mamy hierarchię danych w jednej tabeli, na przykład struktury organizacyjnej lub relacji między danymi. Pozwala nam na skomplikowane zapytania, które wykorzystują powiązania między rekordami w tej samej tabeli.

Typ danych złożonych

Typ danych złożony (composite data type) to typ danych, który pozwala na grupowanie i przechowywanie innych typów danych jako pojedynczej jednostki. Najczęściej używane typy złożone to rekordy (record) i tablice (array).

1. Rekordy (record):

- Rekord to struktura danych, która może zawierać wiele pól o różnych typach danych.
- Definicja rekordu wymaga zdefiniowania nazw pól i odpowiadających im typów danych.
- Przykład składni w PostgreSQL:

```
CREATE TYPE person _type AS (
 first_name VARCHA R(50),
 last_name VARCHA R(50),
 age INT
);
```

- Przykład użycia rekordu:

```
DECLARE
  person person _type;
BEGIN
```

Typ danych złożonych (cont)

```
  person.fi rst _name := 'John';
  person.la st_name := 'Doe';
  person.age := 30;
  ...
END;
```

2. Tablice (array):

- Tablica to struktura danych, która pozwala na przechowywanie wielu wartości tego samego typu w jednym polu.
- Definicja tablicy wymaga określenia typu danych elementów tablicy.
- Przykład składni w PostgreSQL:

```
CREATE TYPE colors AS VARCHA R(20) ARRAY;
```

- Przykład użycia tablicy:

```
DECLARE
  color_list colors := ARRAY[ 'red', 'green', 'blue']
;
  ...
END;
```

Warto zauważyć, że składnia i obsługiwane typy złożone mogą się różnić w zależności od konkretnego systemu zarządzania bazą danych. Przed użyciem konkretnego typu złożonego zaleca się zapoznanie się z dokumentacją systemu bazodanowego, aby uzyskać dokładne informacje na temat składni, zachowania i dostępnych funkcji dla danego typu złożonego.

DECODE - mapowanie wartości

Funkcja `DECODE` w języku SQL jest często używana do mapowania wa warunków. Można to zakwalifikować jako operację warunkowego przyp wartość wyrażenia z zestawem wartości i zwraca odpowiadający wynik Poniżej przedstawiam kilka zastosowań funkcji `DECODE`:

1. Mapowanie wartości:

```
SELECT
    column _name,
    DECODE (co lum n_name, value1, result1, value2, r
value
FROM table_ name;
```

Przykład:

```
SELECT
    produc t_name,
    DECODE (ca teg ory_id, 1, 'Elect ron ics', 2, 'C
FROM products;
```

2. Przypisanie wartości na podstawie warunków:

```
SELECT
    column _name,
    DECODE (co ndi tion, value1, result1, value2, res
FROM table_ name;
```

Przykład:

```
SELECT
    order_ date,
```

DECODE - mapowanie wartości (cont)

```
    DECODE (SI GN( tot al_ amount - 100), 1, 'Above', -1, 'Below','Equal') AS st
atus
FROM orders;
```

Funkcja `DECODE` jest często używana w starszych systemach baz danych, takich jak Oracle. W nowoczesnych bazach danych istnieją również inne konstrukcje, takie jak instrukcja `CASE`, które zapewniają bardziej elastyczną i czytelną składnię do realizacji podobnych operacji warunkowych.

Czym są sekwencje?

Sequences (sekwencje) w bazach danych są obiektami służącymi do generowania unikalnych numerów sekwencyjnych. Są one często używane do automatycznego generowania wartości kluczy głównych lub innych unikalnych identyfikatorów w tabelach.

Główne cechy sekwencji to:

1. Unikalność: Sekwencje zapewniają unikalność generowanych wartości. Każda wygenerowana wartość jest różna od poprzednich i następnych.
2. Bezstanowość: Sekwencje są bezstanowe, co oznacza, że nie przechowują żadnych informacji o poprzednio wygenerowanych wartościach. Przy każdym wywołaniu sekwencji generowana jest kolejna wartość zgodnie z zdefiniowanym przez nas krokiem.
3. Niezależność transakcji: Wywołania sekwencji są niezależne od transakcji. Oznacza to, że wartości sekwencji są generowane niezależnie od innych operacji wykonywanych w ramach transakcji.

Przykład tworzenia i użycia sekwencji w języku SQL (na przykładzie Oracle):

```
-- Tworzenie sekwencji
CREATE SEQUENCE seq_em plo yee_id
    START WITH 1
    INCREMENT BY 1
    NOCACHE;
-- Wykorz ystanie sekwencji do genero wania wartoś
ci
INSERT INTO employees (emplo yee_id, employ ee_ na
me)
VALUES (seq_e mpl oye e_i d.N EXTVAL, 'John Doe');
```

W powyższym przykładzie tworzona jest sekwencja `seq_em plo y ee_id`, która rozpoczyna się od 1 i inkrementuje wartość o 1 przy każdym wywołaniu. Następnie sekwencja jest wykorzystywana do generowania wartości klucza głównego podczas wstawiania danych do tabeli `employees`.

Sekwencje są dostępne w różnych systemach baz danych, takich jak Oracle, PostgreSQL, czy SQL Server. Składnia i opcje tworzenia sekwencji mogą się nieco różnić w zależności od konkretnego systemu bazodanowego. Należy sprawdzić dokumentację swojego systemu baz danych, aby dowiedzieć się więcej o tworzeniu i używaniu sekwencji.

SKŁADNIA

W języku SQL, składnia zapytań zazwyczaj przyjmuje postać:

SELECT - określa, które kolumny mają zostać zwrócone lub jakie wyrażenia mają zostać obliczone.

FROM - wskazuje, z jakich tabel lub widoków mają zostać pobrane dane.

WHERE - określa warunki filtrujące wiersze, które mają zostać zwrócone na podstawie określonych kryteriów.

GROUP BY - grupuje wyniki według określonych kolumn.

HAVING - filtruje grupy na podstawie warunków.

ORDER BY - sortuje wyniki według określonych kolumn w określonej kolejności.

LIMIT/OFFSET (opcjonalne) - ogranicza liczbę zwracanych wierszy lub przesuwa wyniki o określoną liczbę wierszy.

SKŁADNIA (cont)

Oczywiście, nie wszystkie klauzule muszą być obecne w każdym zapytaniu. Składnia i kolejność klauzul zależą od konkretnego rodzaju zapytania i wymagań.

Przykładowa składnia zapytania SELECT może wyglądać tak:

```
SELECT kolumny
FROM tabela
WHERE warunki
GROUP BY kolumny
HAVING warunki
ORDER BY kolumny
LIMIT liczba _wi erszy
OFFSET przesu niecie;
```

Ważne jest przestrzeganie poprawnej składni SQL, aby zapewnić poprawne wykonanie zapytań i otrzymanie oczekiwanych wyników.

Numeryczne typy danych

INTEGER	Reprezentuje liczby całkowite o ograniczonym zakresie	43
BIGINT	Reprezentuje duże liczby całkowite.	9876543210

Numeryczne typy danych (cont)

DECIMAL(p, s) lub NUMERIC(p, s)	Reprezentuje liczby zmiennoprzecinkowe o precyzji p (liczba całkowita) i skali s (liczba miejsc po przecinku).	12.345
FLOAT(p)	Reprezentuje liczby zmiennoprzecinkowe o podwójnej precyzji.	3.14159
REAL	Reprezentuje liczby zmiennoprzecinkowe o jednostronnej precyzji.	2.71828
SMALLINT	Reprezentuje małe liczby całkowite o ograniczonym zakresie.	123
NUMERIC(p, s)	Reprezentuje liczby zmiennoprzecinkowe o precyzji p i skali s.	9876.54321
DOUBLE PRECISION	Reprezentuje liczby zmiennoprzecinkowe o podwójnej precyzji.	1.2345678-9012345

Numeryczne typy danych (cont)

DEC	Reprezentuje liczby zmiennoprzecinkowe o precyzji i skali dostosowanej do konkretnego systemu bazodanowego.	456.789
NUM	Reprezentuje liczby zmiennoprzecinkowe o precyzji i skali dostosowanej do konkretnego systemu bazodanowego.	0.1234

Tekstowe typy danych

CHAR(n)	Reprezentuje stałą długość łańcucha znakowego o rozmiarze n.	'Hello'
VARCHAR(n)	Reprezentuje zmienną długość łańcucha znakowego o maksymalnym rozmiarze n.	'OpenAI'
TEXT	Reprezentuje łańcuch znakowy o zmiennej długości, bez określonego maksymalnego	'Lorem ipsum dolor sit amet...'

Tekstowe typy danych (cont)

NCHAR(n)	Reprezentuje stałą długość łańcucha znakowego w formacie Unicode o rozmiarze n.	'Привет'
NVARCHAR(n)	Reprezentuje zmienną długość łańcucha znakowego w formacie Unicode o maksymalnym rozmiarze n.	'こんにちは'
CLOB	Reprezentuje dużą ilość tekstu o zmiennej długości.	Długi łańcuch znakowy zawierający wiele paragrafów lub rozległe dane tekstowe.
BLOB	Reprezentuje duże binarne dane, takie jak obrazy, dźwięk, wideo itp.	Dane binarne reprezentujące plik graficzny JPEG.
ENUM	Reprezentuje zestaw wartości tekstowych, z których można wybrać jedną.	'Male', 'Female', 'Other'

Tekstowe typy danych (cont)

SET	Reprezentuje zbiór wartości tekstowych, z których można wybrać więcej niż jedną.	'Red', 'Green', 'Blue'

Data i czas

DATE	Reprezentuje datę (bez czasu) w formacie 'RRRR-MM-DD'	'2023-07-04'
TIME	Reprezentuje czas (bez daty) w formacie 'HH:MI:SS'	'12:34:56'
DATETIME lub TIMESTAMP:	Reprezentuje datę i czas w formacie 'RRRR-MM-DD HH:MI:SS'	'2023-07-04 12:34:56'
YEAR	Reprezentuje rok w formacie 'RRRR'	'2023'
INTERVAL	Reprezentuje pewien przedział czasu, np. ilość dni, godzin, minut itp.	INTERVAL '1 day' (reprezentuje 1 dzień)

Data i czas (cont)

TIME WITH TIME ZONE	Reprezentuje czas wraz z informacją o strefie czasowej w formacie 'HH:MI:SS +/-HH:MI'	'12:34:56 +03:00' (czas w strefie czasowej GMT+03:00)
TIMESTAMP WITH TIME ZONE	Reprezentuje datę i czas wraz z informacją o strefie czasowej w formacie 'RRRR--MM-DD HH:MI:SS +/-HH:MI'	'2023-07-04 12:34:56 +03:00' (data i czas w strefie czasowej GMT+03:00)

Pozostałe typy danych

BOOLEAN	Reprezentuje wartość logiczną true lub false	TRUE
BINARY	Reprezentuje dane binarne o stałej długości	01101001
VARBINARY	Reprezentuje dane binarne o zmiennej długości	11001100
BIT	Reprezentuje pojedynczy bit o wartości 0 lub 1	1
UUID	Reprezentuje unikalny identyfikator, często używany do identyfikacji rekordów w tabelach	'550e8400-e2-9b-41d4-a716-446655440000'

Pozostałe typy danych (cont)

XML	Reprezentuje dane w formacie XML	'<person><name>-John</name><age->30</age></pers-on>'
JSON	Reprezentuje dane w formacie JSON (JavaScript Object Notation)	'{"name": "John", "-age": 30}'
GEOMETRY	Reprezentuje dane geometrii przestrzennej, takie jak punkty, linie, poligony itp.	POINT(1 2)
ARRAY	Reprezentuje tablicę wartości jednego typu danych	[1, 2, 3, 4]

Count

COUNT(*)	Zlicza wszystkie wiersze w wyniku zapytania, niezależnie od wartości w poszczególnych kolumnach	`SELECT COUNT(*) FROM Customers;`
COUNT(1)	Zlicza wszystkie wiersze w wyniku zapytania, używając stałej wartości	`SELECT COUNT(1) FROM Customers;`

Count (cont)

COUNT(column-_name)	Zlicza liczbę niepustych wartości w określonej kolumnie	`SELECT COUNT( Qua ntity) FROM Orders;`
COUNT(DIS-TINCT column-_name)	Zlicza liczbę unikalnych niepustych wartości w określonej kolumnie	`SELECT COUNT( DISTINCT Custom erID) FROM Orders;`
COUNT(expres-sion)	Zlicza liczbę niepustych wartości zwracanych przez określone wyrażenie	`SELECT COUNT(CASE WHEN Quantity > 10 THEN 1 END) FROM Orders;`
COUNT(DIS-TINCT expres-sion)	Zlicza liczbę unikalnych niepustych wartości zwracanych przez określone wyrażenie.	`SELECT COUNT( DIS TINCT CONCA T( FirstName, LastName )) FROM Customers;`
IS NOT NULL	Zliczanie niepustych wartości w kolumnie	`SELECT COUNT(*) FROM Customers WHERE City IS NOT NULL;`

Cheatography

SQL Cheat Sheet

by sigeud via cheatography.com/188214/cs/39451/

Count (cont)

IS NULL	Zliczanie wartości NULL w kolumnie	`SELECT COUNT(*) FROM Customers WHERE City IS NULL;`

String functions

CONCAT()	Łączy dwa lub więcej łańcuchów znakowych	`SELECT CONCAT (Fi rst Name, ' ', LastNam e) AS FullName FROM Customers;`
LENGTH() / LEN()	Zwraca długość łańcucha znakowego	`SELECT LENGTH (Fi rst Name) AS NameLength FROM Customers;`
UPPER()	Konwertuje łańcuch znakowy na wielkie litery	`SELECT UPPER( Fir stName) AS UpperName FROM Customers;`
LOWER()	Konwertuje łańcuch znakowy na małe litery	`SELECT LOWER( Las tName) AS LowerName FROM Customers;`

String functions (cont)

SUBSTR-ING()	Wyodrębnia podłańcuch znakowy z danego łańcucha na podstawie określ-onego indeksu i długości	`SELECT SUBSTR ING (De scr ipti AS Substr ingDesc FROM Products;`
REPLACE()	Zamienia wszystkie wystąpienia określ-onego podłań-cucha na inny podłańcuch w łańcuchu	`SELECT REPLAC E(D esc rip tion w') AS Update dDesc FROM Products;`
TRIM() / LTRIM() / RTRIM()	Usuwa początkowe i końcowe białe znaki z łańcucha	`SELECT TRIM(F irs tName) AS Trimme dName FROM Customers;`
LEFT()	Zwraca określoną liczbę znaków z lewej strony łańcucha	`SELECT LEFT(F irs tName, 3) AS LeftChars FROM Customers;`

String functions (cont)

RIGHT()	Zwraca określoną liczbę znaków z prawej strony łańcucha	`SELECT RIGHT( Las tName, 2) AS RightChars FROM Customers;`
SPLIT_PART(string, delimiter, position)	Dzieli podany ciąg znaków na części na podstawie określonego separatora i zwraca część o określonej pozycji	`SELECT SPLIT_ PAR T(' Joh n,D oe,42', ',', 2);` -- 'Doe'
SUBSTRING(string FROM start [FOR length])	Zwraca podciąg znaków z określonego ciągu na podstawie podanego początkowego indeksu i, opcjonalnie, długości.	`SELECT SUBSTR ING ('H ello, World', 1, 5); -- 'Hel lo'`

String functions (cont)

INITCAP(string)	Zamienia pierwszą literę każdego wyrazu w ciągu na wielką literę, a pozostałe litery na małe litery.	`SELECT INITCA P(' hello world'); -- 'Hello World'`
REVERSE(string)	Odwraca kolejność znaków w podanym ciągu.	`SELECT REVERS E(' Hel lo'); -- 'olleH'`

Number funcktions

ABS()	Zwraca wartość bezwzględną liczby	`SELECT ABS(-10) AS Absolu teV alue;`
ROUND()	Zaokrągla liczbę do określonej liczby miejsc po przecinku	`SELECT ROUND( 3.1 4159, 2 ) AS Rounde dNu mber;`
CEILING()	Zwraca najmniejszą liczbę całkowitą większą lub równą danej liczbie	`SELECT CEILIN G(4.25) AS Ceilin gNu mber;`
FLOOR()	Zwraca największą liczbę całkowitą mniejszą lub równą danej liczbie	`SELECT FLOOR( 4.75) AS FloorN umber;`
SQRT()	Zwraca pierwiastek kwadratowy z liczby	`SELECT SQRT(25) AS Square Root;`

Number funcktions (cont)

POWER()	Podnosi daną liczbę do określonej potęgi	`SELECT POWER(2, 3) AS PowerR esult;`
MOD()	Zwraca resztę z dzielenia jednej liczby przez drugą	`SELECT MOD(10, 3) AS Modulo Result;`
RAND()	Zwraca losową liczbę z zakresu od 0 do 1	`SELECT RAND() AS Random Number;`
SIGN()	Zwraca znak liczby (-1 dla liczby ujemnej, 0 dla zera, 1 dla liczby dodatniej)	`SELECT SIGN(-15) AS SignNu mber;`
RANDOM()	Zwraca losową liczbę z określonego zakresu	`SELECT RANDOM() AS Random Number;`
PI()	Zwraca wartość liczby π (pi)	`SELECT PI() AS PiValue;`
LOG()	Oblicza logarytm o podstawie 10 z danej liczby	`SELECT LOG(100) AS Logarithm;`
LN()	Oblicza logarytm naturalny (o podstawie e) z danej liczby	`SELECT LN(2.7 1828) AS Natura lLo gar ithm ;`
EXP()	Oblicza wartość wykładniczą liczby (e) podniesionej do danej potęgi	`SELECT EXP(1) AS Expone nti alV alue ;`

Number funcktions (cont)

GREATEST()	Zwraca największą wartość spośród podanych argumentów	`SELECT GREATE ST(5, 8, 2) AS MaxValue;`
LEAST()	Zwraca najmniejszą wartość spośród podanych argumentów	`SELECT LEAST(5, 8, 2) AS MinValue;`

Date functions

CURRENT_DATE()	Zwraca aktualną datę	`SELECT CURREN T_D ATE() AS Curren tDate;`
CURRENT_TIME()	Zwraca aktualny czas	`SELECT CURREN T_T IME() AS Curren tTime;`
CURRENT_TIMESTAMP()	Zwraca aktualną datę i czas	`SELECT CURREN T_T IME STAMP() AS Curren tDa teTime;`
DATE()	Zwraca tylko datę z wartości daty i czasu	`SELECT DATE(' 202 3-07-04 10:3 AS DateOnly;`
EXTRACT()	Wyodrębnia określony komponent z daty i czasu, takie jak rok, miesiąc, dzień, godzina itp.	`SELECT EXTRAC T(YEAR FROM '202 ) AS Extrac ted Year;`

Date functions (cont)

DATEADD()	Dodaje określoną wartość do daty	`SELECT DATEAD D(DAY, 7, '2023- 07-04') AS AddedDate;`
DATEDIFF()	Oblicza różnicę między dwiema datami w określonej jednostce (np. dni, miesiące, lata)	`SELECT DATEDI FF(DAY, '2023- 07-01', '2023- 07-10') AS DateDi ffe rence;`
DATE_F-ORMAT()	Formatuje datę według określ-onego formatu	`SELECT DATE_F ORM AT( '20 23- 07-04', '%Y-%m -%d' ) AS Format ted Date;`
DATE_P-ART()	Zwraca wartość określonej części daty i czasu, takiej jak rok, miesiąc, dzień, godzina itp.	`SELECT DATE_P ART ('y ear', '2023- 07-04') AS Year;`
DATE_T-RUNC()	Skraca datę do określonej jednostki, np. do miesiąca, roku itp.	`SELECT DATE_T RUN C(' month', '2023- 07-04') AS Trunca ted Date;`
NOW()	Zwraca aktualną datę i czas	`SELECT NOW() AS Curren tDa teTime;`

Date functions (cont)

SYSDATE	Zwraca aktualną datę i czas	`SELECT SYSDATE AS Curren tDa teTime;`
HOW TIMEZONE	Zwraca bieżącą strefę czasową ustawioną w bazie danych	`SHOW TIMEZONE;`
TIME	Typ danych TIME reprezentuje wartość czasu bez daty	`SELECT TIME '10:30:45' AS TimeValue;`

date_part('unit', date) - unit

day	Day of the month (1 to 31)
decade	Year divided by 10
dow	Day of the week (0=Sunday, 1=Monday, 2=Tuesday, ... 6=Saturday)
doy	Day of the year (1=first day of year, 365/366=last day of the year, depending if it is a leap year)
epoch	Number of seconds since '1970-01-01 00:00:00 UTC', if date value. Number of seconds in an interval, if interval value
hour	Hour (0 to 23)
isodow	Day of the week (1=Monday, 2=Tuesday, 3=Wednesday, ... 7=Sunday)

date_part('unit', date) - unit (cont)	
isoyear	ISO 8601 year value (where the year begins on the Monday of the week that contains January 4th)
minute	Minute (0 to 59)
month	Number for the month (1 to 12), if date value. Number of months (0 to 11), if interval value
quarter	Quarter (1 to 4)
second	Seconds (and fractional seconds)
timezone	ime zone offset from UTC, expressed in seconds
timezo-ne_hour	Hour portion of the time zone offset from UTC
timezo-ne_-minute	Minute portion of the time zone offset from UTC
week	Number of the week of the year based on ISO 8601 (where the year begins on the Monday of the week that contains January 4th)
year	Year as 4-digits

https://www.techonthenet.com/postgresql/functions/date_part.php

null functions		
IS NULL	Sprawdza, czy określona wartość jest NULL-em	`SELECT * FROM tabela WHERE kolumna IS NULL;`
IS NOT NULL	Sprawdza, czy określona wartość nie jest NULL-em	`SELECT * FROM tabela WHERE kolumna IS NOT NULL;`
COALESCE()	Zwraca pierwszą nie-NULL-ową wartość z listy wartości	`SELECT COALES CE( kol umna1 , kolumna2, kolumna3) AS NonNul lValue FROM tabela;`
NULLIF()	Zwraca NULL, jeśli dwa wyrażenia są równe, w przeciwnym razie zwraca pierwsze wyrażenie	`SELECT NULLIF (ko lumna1, 0 ) AS Result FROM tabela;`
NVL()	Zwraca drugie wyrażenie, jeśli pierwsze jest NULL-em	`SELECT NVL(ko lumna1, 'Brak wartości') AS Result FROM tabela;`

null functions (cont)

IFNULL()	Zwraca drugie wyrażenie, jeśli pierwsze jest NULL-em	`SELECT IFNULL (ko lumna1 , 'Brak wartości') AS Result FROM tabela;`

Funkcje agregujące

COUNT()	Zlicza liczbę wierszy lub wartości w danej kolumnie.	`SELECT COUNT(*) AS TotalRows FROM tabela;`
SUM()	Oblicza sumę wartości w danej kolumnie numerycznej.	`SELECT SUM(ko lumna) AS TotalSum FROM tabela;`
AVG()	Oblicza średnią wartość w danej kolumnie numerycznej.	`SELECT AVG(ko lumna) AS Averag eValue FROM tabela;`
MIN()	Zwraca najmniejszą wartość w danej kolumnie.	`SELECT MIN(ko lumna) AS MinValue FROM tabela;`

Funkcje agregujące (cont)

MAX()	Zwraca największą wartość w danej kolumnie.	`SELECT MAX(ko lumna) AS MaxValue FROM tabela;`
GROUP_CONCAT (MySQL, MariaDB)	Konkatenacja wartości napisowych z grupy w jedną wartość napisową, rozdzieloną separatorem.	`SELECT GROUP_ CON CAT (na zwa _k lumny SEPARATOR ', ') AS Concat ena ted Values FROM tabela GROUP BY kolumna;`
LISTAGG (Oracle)	Konkatenacja wartości napisowych z grupy w jedną wartość napisową, rozdzieloną separatorem.	`SELECT LISTAG G(n azw a_k olumny ', ') WITHIN GROUP (ORDER BY kolumna) AS Concat ena ted Values FROM tabela;`
GROUP BY	Grupuje wyniki zapytania według wartości określonych kolumn.	`SELECT kolumna, COUNT(*) FROM tabela GROUP BY kolumna;`

Funkcje agregujące (cont)

GROUP BY ROLLUP	pozwala na generowanie zestawu wyników hierarchicznych w oparciu o różne poziomy podsumowania, generuje wyniki dla wszystkich kombinacji wartości w kolumnach podsumowania, tworząc hierarchię podsumowań	`SELECT kolumna1, kolumna2, SUM(wa rtosc) AS Suma FROM tabela GROUP BY ROLLUP (ko lumna1, kolumna2);`
GROUP BY CUBE	generuje wszystkie możliwe kombinacje wartości w kolumnach podsumowania, tworząc tzw. kostkę (cube)	`SELECT kolumna1, kolumna2, SUM(wa rtosc) AS Suma FROM tabela GROUP BY CUBE(k olu mna1, kolumna2);`

Funkcje agregujące (cont)

HAVING	służy do filtrowania wyników zapytania po grupowaniu, na podstawie warunków logicznych	`SELECT kolumna, SUM(wa rtosc) AS Suma FROM tabela GROUP BY kolumna HAVING SUM(wa rtosc) > 10 0`

GROUP BY ROLLUP vs GROUP BY CUBE

Przedstawmy porównanie klauzul GROUP BY ROLLUP i CUBE na pod
uproszczonej tabeli zawierającej dane o sprzedaży produktów w różnyc
Tabela "Sales":

```
+---------+------------+----------+
| Product | Region | Quantity |
+---------+------------+----------+
| A | North | 10 |
| A | South | 5 |
| B | North | 8 |
| B | South | 12 |
+---------+------------+----------+
```

Group by ROLLUP:

```
SELECT Product, Region, SUM(Qu antity) AS TotalQ uant
BY ROLLUP (Pr oduct, Region);
```

Wyniki zapytania:

GROUP BY ROLLUP vs GROUP BY CUBE (cont)

```
+---------+------------+---------------+
| Product | Region | TotalQuantity |
+---------+------------+---------------+
| A | North | 10 |
| A | South | 5 |
| A | NULL | 15 |
| B | North | 8 |
| B | South | 12 |
| B | NULL | 20 |
| NULL | NULL | 35 |
+---------+------------+---------------+
```

Klauzula GROUP BY ROLLUP generuje zestaw wyników z hierarchią podsumowań, w którym uwzględnia podsumowanie dla poszczególnych wartości w kolumnach Product i Region oraz całkowite podsumowanie dla każdej kolumny i całkowite podsumowanie.

Group by CUBE:

```
SELECT Product, Region, SUM(Qu antity) AS TotalQ uantity GROUP BY CUBE(P roduct, Region);
```

Wyniki zapytania:

```
+---------+------------+---------------+
| Product | Region | TotalQuantity |
+---------+------------+---------------+
| A | North | 10 |
| A | South | 5 |
| A | NULL | 15 |
| B | North | 8 |
| B | South | 12 |
| B | NULL | 20 |
| NULL | North | 18 |
| NULL | South | 17 |
| NULL | NULL | 35 |
+---------+------------+---------------+
```

Klauzula GROUP BY CUBE generuje zestaw wyników, w którym uwzględnia wszystkie możliwe kombinacje wartości w kolumnach Product i Region, włączając podsumowania dla poszczególnych kolumn oraz całkowite podsumowanie.

Podsumowanie, w którym GROUP BY ROLLUP generuje zestaw wyników z hierarchią podsumowań, w którym uwzględnia podsumowania dla poszczególnych kolumn i ich kombinacji.

GROUP BY CUBE generuje zestaw wyników, który zawiera wszystkie możliwe kombinacje podsumowań dla kolumn, włączając podsumowania dla poszczególnych kolumn.

Warto zauważyć, że w obu przypadkach wyniki zawierają dodatkowe wiersze i kolumny, które reprezentują ogólne podsumowania

Podzapytania (subquery)

Podzapytanie (ang. subquery) w SQL to zapytanie umieszczone wewnątrz innego zapytania. Może być używane w różnych częściach zapytania, takich jak klauzula SELECT, FROM, WHERE, HAVING lub JOIN, w celu uzyskania dodatkowych informacji lub filtrowania danych.

Oto kilka ważnych informacji na temat podzapytań:

Podzapytania (subquery) (cont)

Cel podzapytań: Podzapytania służą do uzyskania danych z innej tabeli lub wyniku innych zapytań wewnątrz głównego zapytania. Mogą dostarczać dodatkowe informacje, filtrować wyniki, dokonywać obliczeń lub dostarczać wyniki dla innych części zapytania.
Składnia: Podzapytania są umieszczane wewnątrz nawiasów okrągłych lub kwadratowych i są traktowane jako oddzielne zapytania. Mogą być umieszczone w różnych częściach zapytania, w zależności od potrzeb.
Przykład:

```
SELECT column1
FROM table1
WHERE column2 IN (SELECT column3 FROM table2);
```

Związki z innymi zapytaniami: Podzapytania mogą być łączone z innymi zapytaniami przy użyciu operatorów takich jak IN, EXISTS, ANY, ALL, czy nawet z innymi podzapytaniami, w celu tworzenia bardziej zaawansowanych zapytań.
Przykład:

```
SELECT column1
FROM table1
WHERE column2 IN (SELECT column3 FROM table2 WHERE column4 > 100);
```

Wydajność: Podzapytania mogą mieć wpływ na wydajność zapytań, zwłaszcza gdy są używane wewnątrz klauzuli WHERE lub HAVING. Ważne jest optymalizowanie zapytań zawierających podzapytania, aby uniknąć zbędnych obliczeń i nadmiernego przetwarzania danych.

Podzapytania (subquery) (cont)

Podzapytania są potężnym narzędziem w SQL, pozwalającym na bardziej zaawansowane manipulowanie danymi i tworzenie bardziej przejrzystych zapytań. Ich zastosowanie zależy od konkretnego przypadku i wymagań zapytania, dlatego warto zapoznać się z nimi i zrozumieć ich składnię i możliwości.

Podzapytania CTE

Podzapytania CTE (Common Table Expressions) to narzędzie w SQL, umożliwia tworzenie tymczasowych, nazwanych zestawów wyników, kt następnie wykorzystać w głównym zapytaniu. CTE są szczególnie przy przypadku bardziej skomplikowanych zapytań, gdzie wymagane jest wi użycie tego samego podzapytania.
Składnia CTE wygląda następująco:

```
WITH nazwa_cte (kolumna1, kolumna2, ...)
AS (
    SELECT kolumna1, kolumna2, ...
    FROM tabela
    WHERE warunek
)
```

Przykład:

```
WITH sales_cte (produ ct_id, total_ sales)
AS (
    SELECT produc t_id, SUM(qu antity * price) AS tot
es
    FROM sales
    GROUP BY product_id
)
```

Podzapytania CTE (cont)

```
SELECT produc t_id, total_ sales
FROM sales_cte
WHERE total_ sales > 1000;
```

W tym przykładzie tworzymy CTE o nazwie "sales_cte", która zawiera wyniki sumy sprzedaży dla każdego produktu. Następnie w głównym zapytaniu wybieramy dane z CTE, gdzie wartość sprzedaży jest większa niż 1000.

```
WITH cte1 AS (
    SELECT column1, column2
    FROM table1
    WHERE condition1
),
cte2 AS (
    SELECT column3, column4
    FROM table2
    WHERE condition2
)
SELECT cte1.c olumn1, cte1.c olumn2, cte2.c olumn3, cte2.c ol
umn4
FROM cte1
JOIN cte2 ON cte1.c olumn1 = cte2.c olumn3;
```

W powyższym przykładzie definiujemy dwie CTE: "cte1" i "cte2". CTE "cte1" wybiera kolumny "column1" i "column2" z "table1" spełniające określony warunek. CTE "cte2" wybiera kolumny "column3" i "column4" z "table2" spełniające inny warunek. Następnie wykonujemy złączenie (JOIN) między wynikami obu CTE, używając kolumny "column1" z "cte1" i kolumny "column3" z "cte2".

Podzapytania CTE (cont)

Podzapytania CTE mogą znacznie ułatwić czytelność i zarządzanie bar niami, pozwalając na ich modularyzację i ponowne wykorzystanie.

Klauzula WITH RECURSIVE w języku SQL umożliwia tworzenie rekure być używane do przetwarzania danych hierarchicznych lub grafowych. wanie zapytań z użyciem wyników poprzednich iteracji jako dane wejśc

Składnia ogólna WITH RECURSIVE wygląda następująco:

```
WITH RECURSIVE nazwa (kolumny) AS (
   Zapyta nie _ni e_r eku ren cyjne
   UNION [ALL]
   Zapyta nie _re kur encyjne
)
Zapyta nie _główne
```

Przykład:

Załóżmy, że mamy tabelę "employees" zawierającą dane pracowników kierownicza. Chcemy wykonać rekurencyjne zapytanie, które zwróci ws podlegają danemu kierownikowi (w dowolnym stopniu hierarchii).

```
WITH RECURSIVE employ ee_ hie rarchy (emplo yee_id,
 AS (
   SELECT employ ee_id, full_name, manager_id
   FROM employees
   WHERE employ ee_id = 1 -- Przykł adowy identy fik
   UNION ALL
   SELECT e.empl oye e_id, e.full _name, e.mana ger_
   FROM employees e
```

Podzapytania CTE (cont)

```
   JOIN employ ee_ hie rarchy eh ON e.mana ger_id = eh.emp
ee_id
)
SELECT *
FROM employ ee_ hie rarchy;
```

W tym przykładzie używamy WITH RECURSIVE do stworzenia rekurencyjnego zapytania o nazwie "employee_hierarchy". W początkowym zapytaniu wybieramy dane kierownika o określonym identyfikatorze. Następnie używamy operatora UNION ALL i dołączamy zapytanie rekurencyjne, które odwołuje się do "employee_hierarchy" i łączy pracowników, których identyfikator kierownika jest równy identyfikatorowi z poprzedniej iteracji.
W rezultacie zapytanie zwróci wszystkich pracowników, którzy są podlegli danemu kierownikowi, bez względu na poziom hierarchii.

Losowa próbka rekordów

Aby pobrać losową próbkę rekordów z tabeli, można skorzystać z różnych technik, w zależności od używanego systemu bazodanowego. Oto kilka przykładów:
- Przykład dla systemu PostgreSQL:

```
SELECT *
FROM table_name
ORDER BY RANDOM()
LIMIT 10;
```

W powyższym przykładzie używamy funkcji RANDOM() w instrukcji ORDER BY, aby przemieszać rekordy w tabeli, a następnie ograniczamy wyniki do 10 przy użyciu LIMIT, co daje nam losową próbkę 10 rekordów.
Przykład dla systemu MySQL:

Losowa próbka rekordów (cont)

```
SELECT *
FROM table_name
ORDER BY RAND()
LIMIT 10;
```

W przypadku MySQL używamy funkcji RAND() w instrukcji ORDER BY do przemieszania rekordów, a następnie ograniczamy wyniki do 10 przy użyciu LIMIT.
Przykład dla systemu Microsoft SQL Server:

```
SELECT TOP 10 *
FROM table_name
ORDER BY NEWID();
```

W przypadku SQL Servera możemy użyć funkcji NEWID() w instrukcji ORDER BY, która generuje unikalne identyfikatory GUID, co daje nam losowe sortowanie rekordów. Następnie używamy TOP do ograniczenia wyników do 10 rekordów.
- WHERE RANDOM() < 0.01: Ta metoda wykorzystuje funkcję RANDOM() w warunku WHERE, aby wybrać tylko te rekordy, których wartość losowa jest mniejsza niż 0.01. Można dostosować wartość 0.01, aby uzyskać różne wielkości próbek.
Przykład:

```
SELECT *
FROM table_name
WHERE RANDOM() < 0.01;
```

- TABLESAMPLE SYSTEM(1): Ta metoda wykorzystuje klauzulę TABLESAMPLE w zapytaniu, która umożliwia losowe pobranie próbki rekordów na podstawie procentowego udziału w tabeli.
Przykład:

Losowa próbka rekordów (cont)

```
SELECT *
FROM table_name TABLES AMPLE SYSTEM(1);
```

W powyższym przykładzie SYSTEM(1) oznacza, że zostanie pobrana próbka o wielkości około 1% całej tabeli. Można dostosować wartość procentową, aby uzyskać różne rozmiary próbek.

PIVOT

Pivot w SQL to operacja, która umożliwia transformację danych w pionowy format (kolumny w wiersze) na poziomy format (wiersze w kolumny). Jest to przydatne, gdy chcemy dokonać agregacji danych wokół określonej kolumny lub zestawić dane w bardziej zrozumiały sposób.

Operacja Pivot wymaga zdefiniowania, które kolumny zostaną transp-onowane na wartości kolumn w nowej tabeli wynikowej. Oto przykładowe zapytanie wykorzystujące operację Pivot:

```
SELECT *
FROM (
  SELECT category, product, quantity
  FROM sales
) AS src
 PIVOT (
  SUM(qu antity)
 FOR product IN ([Prod uctA], [Produ ctB], [Produ ctC
])
) AS pivot_ table;
```

PIVOT (cont)

W powyższym przykładzie mamy tabelę "sales" zawierającą kolumny "c ntity". Chcemy dokonać transpozycji wartości kolumny "product" na nov wynikowej. W zapytaniu definiujemy listę produktów, dla których chcem ([ProductA], [ProductB], [ProductC]). Następnie używamy funkcji agregu SUM) do obliczenia wartości dla każdego produktu w ramach danej kat Takie zapytanie Pivot zwróci tabelę wynikową, w której poszczególne p tawione jako osobne kolumny, a wartości będą agregowane dla każdej

category | ProductA | ProductB | ProductC

CategoryA | 10 | 5 | 3

CategoryB | 8 | 2 | 6

Warto zauważyć, że składnia zapytania Pivot może się różnić w zależn systemu bazodanowego. Przykład powyżej jest ogólnym przykładem, a dokumentację systemu bazodanowego, którego używasz, aby poznać s dotyczącą operacji Pivot w danym systemie.

a)
```
SELECT *
FROM crosst ab('SQL tworzący tabelę w postaci tekstu'
AS nazwa_ tab eli (de finicja kolumn wynikowych tabel
```

b)
```
SELECT *
FROM crosst ab( 'select jezyk: :text, stan_w nio sku
:nu meric
FROM wnioski group by 1, 2')
```

PIVOT (cont)

```
AS final_ res ult (“j ezyk” text, " odr zucony prawni e" numeric, " odr zucony po analiz ie"
 numeric, " zaa kce ptowany przez operat ora " numeric, " wyp łac ony " numeric, " zam knię e
ty " numeric, " akcja sadowa " numeric, " now y" numeric, " analiza zaakce pto wan a " numeri
c, " prz egrany w sadzie " numeric, " wyslany do operat ora " numeric, " odr zucony przez op
erat ora " numeric, " wygrany w sadzie " numeric);
```

Indeksowanie

Indeksy są strukturami danych w bazach danych, które pomagają przyspieszyć wyszukiwanie i sortowanie danych. Są to specjalne obiekty, które są tworzone na jednym lub wielu kolumnach tabeli i umożliwiają szybkie odnajdywanie odpowiednich rekordów na podstawie wartości w tych kolumnach.

Oto kilka podstawowych informacji o indeksach:

Rodzaje indeksów:

Indeksy jednok olu mnowe: tworzone na pojedy nczej kolumnie tabeli.

Indeksy wielok olu mnowe: tworzone na kilku kolumnach tabeli.

Unikalne indeksy: zapewn iają, że wartości w indeks owanych kolumnach są unikalne.

Indeksy skupione (clust ered): określają fizyczną organi zację danych w tabeli na podstawie wartości indeks owanych kolumn.

Indeksy niestr ukture (non-c lus tered): mają oddzielną strukturę od tabeli i zawierają odnośniki do fizycznych miejsc, gdzie dane są przech owy - wane.

Korzyści wynikające z indeks owania:

Szybsze wyszuk iwanie danych: indeksy umożli wiają bezpoś rednie odnale zienie pasujących rekordów, co przysp iesza zapytania wyszuk iwania.

Szybsze sortowanie danych: indeksy ułatwiają sortowanie danych według indeks owanych kolumn.

Zmniej szone obciążenie zapytań: efektywne indeks owanie może zmniejszyć obciążenie na serwerze baz danych poprzez skrócenie czasu wykony wania zapytań.

Indeksowanie (cont)

> Tworzenie i zarządzanie indeksami:

Indeksy można tworzyć podczas tworzenia tabeli lub po jej utworzeniu.

Tworzenie indeksów powinno być odpowiednio przemyślane, uwzględn wykonywane zapytania i typ operacji, które będą wykonywane na dany

Indeksy należy utrzymywać i aktualizować wraz z danymi w tabelach, a ich skuteczność.

Przykład tworzenia indeksu na jednej kolumnie:

```
CREATE INDEX idx_cu sto mer _name ON Customers (custo
e);
```

Przykład tworzenia indeksu na wielu kolumnach:

```
CREATE INDEX idx_cu sto mer _lo cation ON Customers
te);
```

Indeksy są ważnym elementem optymalizacji baz danych, które pomag poprawie wydajności zapytań i operacji na danych. Ważne jest jednak projektowanie i zarządzanie indeksami, aby uniknąć nadmiernego indel które może prowadzić do spadku wydajności podczas modyfikacji dany

Indeksy nie są bezpośrednio widoczne po wywołaniu tabeli, ale mają w wydajność zapytań, szczególnie podczas wyszukiwania, sortowania i ła danych. Przykład poniżej ilustruje, jak indeksy mogą przyspieszyć zapy Mamy tabelę "Customers" z indeksem na kolumnie "customer_name". zapytanie wyszukujące klienta o nazwie "John Smith":

```
SELECT * FROM Customers WHERE custom er_name = 'John
```

Bez indeksu, silnik bazy danych musiałby przeszukać całą tabelę w pos rekordów spełniających warunek. Jeśli tabela ma dużą liczbę rekordów może być wolne.

Indeksowanie (cont)

> Z indeksem na kolumnie "customer_name", silnik bazy danych może skorzystać z indeksu, aby bezpośrednio odnaleźć rekordy o nazwie "John Smith", nie musząc przeszukiwać całej tabeli. Wykonanie zapytania będzie znacznie szybsze.
Indeksy są tworzone dla optymalizacji zapytań i ukryte są wewnętrznie w silniku bazy danych. Nie są one bezpośrednio widoczne jako kolumny lub dane, ale mają wpływ na wydajność zapytań, a wyniki mogą być widoczne w czasie wykonywania zapytań.

łączenie (konkatenacja) stringów

W SQL istnieje kilka sposobów łączenia (konkatenacji) stringów. Oto kilka przykładów:
- Operator konkatenacji (+):
Możesz użyć operatora "+" do połączenia dwóch stringów w jedną wartość. Przykład:

```
SELECT 'Hello' + ' ' + 'World' AS concat ena ted _st ring;
```

Wynik: "Hello World"
- Funkcja CONCAT():
Funkcja CONCAT() pozwala na łączenie wielu stringów w jeden. Przykład:

```
SELECT CONCAT ('H ello', ' ', 'World') AS concat ena ted _st
 ring;
```

Wynik: "Hello World"
- Operator ||:
W niektórych bazach danych, takich jak Oracle, PostgreSQL czy SQLite, można użyć operatora "||" do konkatenacji stringów. Przykład:
SELECT 'Hello' || ' ' || 'World' AS concatenated_string;
Wynik: "Hello World"
- Funkcja CONCAT_WS():

łączenie (konkatenacja) stringów (cont)

Funkcja CONCAT_WS() służy do konkatenacji stringów z użyciem okre
Przykład:

```
SELECT CONCAT _WS(', ', 'John', 'Doe', 'New York') AS
ring;
```

Wynik: "John, Doe, New York"
- Funkcja STUFF() (dla SQL Server):
Funkcja STUFF() w SQL Server umożliwia zamianę fragmentu stringa i używana do łączenia stringów. Przykład:

```
SELECT STUFF( 'Hello World', 6, 0, ', ') AS concat en
```

Wynik: "Hello, World"

Konwersja danych

w zapytaniu SELECT istnieją pewne funkcje konwersji danych, które mogą służyć do przekształcenia wartości jednego typu danych na inny typ danych w wynikach zapytania. Oto kilka przykładów:
- CAST() lub CONVERT() umożliwiają konwersję jednego typu danych na inny. Przykład:

```
SELECT CAST(k olumna AS nowy_typ) AS nowa_k olumna
FROM tabela;
```

lub

```
SELECT CONVER T(n owy _typ, kolumna) AS nowa_k olu
mna
FROM tabela;
```

Należy zastąpić "kolumna" nazwą kolumny, którą chcesz przekonwertować, a "nowy_typ" docelowym typem danych.
*Convert z unicode

```
conver t(name, 'AL32U TF8', 'WE8MS WIN 1252')
```

Konwersja danych (cont)

```
SELECT CONVER T(name, 'UTF8', 'AL32U TF8') FROM nazwa_tabeli
```

- funkcje konwersji typów danych:

W zależności od używanej bazy danych, mogą istnieć specyficzne funkcje konwersji typów danych, takie jak TO_NUMBER(), TO_DATE(), TO_CHAR() itp. Możesz użyć tych funkcji w zapytaniu SELECT, aby przekonwertować wartości na określony typ danych. Przykład:

```
SELECT TO_NUM BER (ko lumna) AS nowa_k olumna
FROM tabela;
```

W powyższym przykładzie używamy funkcji TO_NUMBER(), aby przekonwertować wartość kolumny na typ liczbowy.

-Składnia wyrażenie::typ_danych lub wyrażenie::typ_danych(n) służy do jawnego rzutowania (explicit cast) wartości wyrażenia na określony typ danych. Przykład:

```
SELECT kolumn a:: integer AS nowa_k olumna
FROM tabela;
```

W powyższym przykładzie używamy ::integer, aby przekonwertować wartość kolumny na typ całkowitoliczbowy (integer).

Warto zauważyć, że ten zapis ::typ_danych jest specyficzny dla niektórych baz danych, takich jak PostgreSQL, i nie jest powszechnie obsługiwany we wszystkich bazach danych. Dlatego zawsze warto sprawdzić dokumentację konkretnej bazy danych, aby upewnić się, czy ten zapis jest dostępny i poprawnie obsługiwany w danym systemie bazodanowym.

trim

Funkcja TRIM w języku SQL jest używana do usuwania spacji lub innych określonych znaków z początku i końca ciągu znaków. Funkcja TRIM może być również rozszerzona o specyfikację konkretnych znaków, które mają być usunięte z ciągu.

Składnia ogólna funkcji TRIM jest następująca:

TRIM([[LEADING | TRAILING | BOTH] [characters_to_remove] FROM] string_expression)

Argumenty funkcji TRIM:

- LEADING: Usuwa określone znaki z początku ciągu.
- TRAILING: Usuwa określone znaki z końca ciągu.
- BOTH (domyślne): Usuwa określone znaki zarówno z początku, jak i końca ciągu.
- characters_to_remove (opcjonalny): Określa konkretne znaki, które mają być usunięte. Może to być ciąg znaków lub kolumna zawierająca znaki.
- string_expression: Wyrażenie tekstowe, z którego mają być usunięte znaki.

Przykłady:

Usunięcie spacji z początku i końca ciągu:

```
SELECT TRIM('   Hello World   ');
```

Wynik: 'Hello World'

Usunięcie określonych znaków z początku i końca ciągu:

```
SELECT TRIM('.,?!' FROM '...Hello, World? !...');
```

Wynik: 'Hello, World'

Usunięcie spacji tylko z początku ciągu:

```
SELECT TRIM(L EADING ' ' FROM '   Hello World   ');
```

Wynik: 'Hello World '

Usunięcie spacji tylko z końca ciągu:

trim (cont)

```
SELECT TRIM(T RAILING ' ' FROM '   Hello World   ');
```

Wynik: ' Hello World'

Funkcja TRIM jest przydatna do usuwania zbędnych białych znaków z ciągów, takich jak spacje, które mogą wpływać na porównania, grupowanie lub inne operacje na danych tekstowych.

Warto pamiętać, że dostępność i składnia funkcji TRIM mogą się różnić w zależności od używanej bazy danych. Należy zawsze sprawdzić dokumentację konkretnej bazy danych w celu uzyskania dokładnej składni i zachowania funkcji TRIM.

Różnica między EXISTS a IN

Klauzula EXISTS i IN służą do filtrowania danych w zapytaniu na podstawie wartości w innym podzapytaniu lub zbiorze wartości. Oto różnica między nimi:

1. EXISTS:

- Klauzula EXISTS sprawdza, czy podzapytanie zwraca jakiekolwiek wyniki.
- Jeśli podzapytanie zwraca przynajmniej jeden wiersz, to warunek EXISTS jest spełniony.
- Klauzula EXISTS jest zwykle używana w warunku WHERE, aby sprawdzić, czy istnieją powiązane wiersze w innych tabelach.
- Może być bardziej wydajna niż IN w przypadku dużych zbiorów danych, ponieważ po znalezieniu pierwszego pasującego wiersza, dalsze poszukiwania są przerwane.

Przykład użycia klauzuli EXISTS:

```
SELECT column1, column2, ...
FROM table1
WHERE EXISTS (SELECT column1 FROM table2 WHERE condit ion)
;
```

2. IN:

Różnica między EXISTS a IN (cont)

- Klauzula IN porównuje wartość wyrażenia z zestawem wartości dostar podzapytaniu lub liście wartości.
- Jeśli wartość jest równa jednej z wartości z podzapytania lub listy, war spełniony.
- Klauzula IN jest zwykle używana w warunku WHERE, aby sprawdzić, znajduje się w określonym zestawie.
- Może być łatwiejsza do zrozumienia i zapisu niż EXISTS, ale może by wydajna dla dużych zbiorów danych.

Przykład użycia klauzuli IN:

```
SELECT column1, column2, ...
FROM table1
WHERE column1 IN (SELECT column1 FROM table2 WHERE co
;
```

Podsumowując, klauzula EXISTS sprawdza istnienie przynajmniej jedn pasującego wiersza w podzapytaniu, podczas gdy klauzula IN porównu zestawu wartości z podzapytania lub listy. Wybór między nimi zależy od przypadku i preferencji programisty.

Różnice między DELETE a TRUNCATE

Różnica między operacjami DELETE i TRUNCATE dotyczy sposobu usuwania danych w bazie danych. Oto główne różnice między nimi:

DELETE:

- DELETE jest operacją DML (Data Manipulation Language), która służy do usuwania jednego lub więcej wierszy z tabeli.
- Operacja DELETE jest rejestrowana w dzienniku transakcji (ang. transaction log), co oznacza, że można cofnąć operację DELETE w ramach transakcji.

Różnice między DELETE a TRUNCATE (cont)

- DELETE można stosować z dodatkowymi warunkami (WHERE), aby precyzyjnie określić, które wiersze mają być usunięte.
- DELETE wywołuje wyzwalacze (triggery) zdefiniowane na tabeli, które mogą wykonywać określone działania przed lub po usunięciu danych.

Przykład użycia DELETE:

```
DELETE FROM tabela WHERE warunek;
```

TRUNCATE:

- TRUNCATE jest operacją DDL (Data Definition Language), która służy do usuwania wszystkich wierszy z tabeli.
- Operacja TRUNCATE nie jest rejestrowana w dzienniku transakcji, co oznacza, że nie można cofnąć operacji TRUNCATE w ramach transakcji.
- TRUNCATE nie używa warunków (WHERE), ponieważ usuwa wszystkie wiersze z tabeli.
- TRUNCATE nie wywołuje wyzwalaczy (triggers) zdefiniowanych na tabeli.

Przykład użycia TRUNCATE:

```
TRUNCATE TABLE tabela;
```

Podsumowując, DELETE jest operacją bardziej precyzyjną, która umożliwia usuwanie wybranych wierszy z tabeli, rejestrowanie w dzienniku transakcji i wywoływanie wyzwalaczy. TRUNCATE jest szybszą operacją, która usuwa wszystkie wiersze z tabeli, nie rejestruje się w dzienniku transakcji i nie wywołuje wyzwalaczy. Wybór między nimi zależy od konkretnego przypadku i wymagań dotyczących operacji usuwania danych.

Usuwanie duplikatów z tabeli

Aby usunąć duplikaty z tabeli, można skorzystać z kombinacji operacji SELECT DISTINCT i INSERT INTO lub operacji DELETE z użyciem klauzuli EXISTS lub CTE (Common Table Expression). Oto kilka przykładów:

Przykład 1: Użycie operacji SELECT DISTINCT i INSERT INTO

```
CREATE TABLE tabela2 AS
SELECT DISTINCT *
FROM tabela;
```

W tym przykładzie tworzymy nową tabelę tabela2, która zawiera tylko unikalne wiersze z tabeli oryginalnej.

Przykład 2: Użycie operacji DELETE z klauzulą EXISTS

```
DELETE FROM tabela a
WHERE EXISTS (
  SELECT 1
  FROM tabela b
  WHERE a.kolumna = b.kolumna
  AND a.id < b.id
);
```

W tym przykładzie usuwamy wiersze, dla których istnieje inny wiersz o tej samej wartości kolumny, ale z niższym identyfikatorem (id).

Przykład 3: Użycie operacji DELETE z wykorzystaniem CTE

```
WITH duplicates AS (
  SELECT kolumna, COUNT(*) AS count
  FROM tabela
```

Usuwanie duplikatów z tabeli (cont)

```
  GROUP BY kolumna
  HAVING COUNT(*) > 1
)
DELETE FROM tabela
WHERE (kolumna) IN (
  SELECT kolumna
  FROM duplicates
);
```

W tym przykładzie tworzymy wspólne wyrażenie tabeli (CTE) o nazwie "duplicates", które identyfikuje kolumny, które mają więcej niż jedno powtórzenie. Następnie wykonujemy operację DELETE, usuwając wiersze, które mają wartości kolumny zawarte w wynikach CTE.

Należy pamiętać, że przed wykonaniem operacji usuwania zawsze warto zrobić kopię zapasową danych lub przetestować zapytanie na kopii testowej bazy danych, aby upewnić się, że operacja usunięcia działa zgodnie z oczekiwaniami i nie spowoduje utraty niepożądanych danych.

Fuzzy Matching

Fuzzy matching, czyli dopasowywanie przybliżone, może być realizowane w języku SQL za pomocą różnych technik. Oto kilka popularnych sposobów realizacji fuzzy matching w SQL:

- Operator LIKE z użyciem symbolu %: Operator LIKE w połączeniu z symbolem % umożliwia dopasowywanie wzorców z użyciem kawałków tekstu. Symbol % reprezentuje dowolną liczbę znaków (również zero znaków). Przykład:

```
SELECT column _name
FROM table_name
WHERE column _name LIKE '%fraza%';
```

Fuzzy Matching (cont)

W powyższym przykładzie, '%fraza%' dopasuje wartości w kolumnie column_name, które zawierają "fraza" gdziekolwiek wewnątrz ciągu.

- Funkcja SOUNDEX: Funkcja SOUNDEX w niektórych bazach danych (np. MySQL, SQL Server) generuje kod dźwiękowy dla podanego wyrażenia. Działa na podstawie podobieństwa dźwiękowego słów. Może być używana do porównywania słów w celu znalezienia podobnych wyników. Przykład:

```
SELECT column _name
FROM table_name
WHERE SOUNDE X(c olu mn_ name) = SOUNDE X(' wyr aż
e nie');
```

W powyższym przykładzie, SOUNDEX(column_name) = SOUNDEX('wyrażenie') porównuje dźwiękowe kody słów w kolumnie column_name z dźwiękowym kodem słowa "wyrażenie".

Funkcje tekstowe: W niektórych bazach danych istnieją specjalne funkcje tekstowe do przeprowadzania dopasowania przybliżonego, takie jak DIFFERENCE w SQL Server. Te funkcje obliczają podobieństwo między dwoma wyrażeniami i zwracają wartość liczbową reprezentującą stopień dopasowania.

- Przykład z użyciem DIFFERENCE w SQL Server:

```
SELECT column _name
FROM table_name
WHERE DIFFER ENC E(c olu mn_ name, 'wyraż enie') >
= 3;
```

W powyższym przykładzie, DIFFERENCE(column_name, 'wyrażenie') >= 3 porównuje podobieństwo między słowami w kolumnie column_name a słowem "wyrażenie". Wartość 3 lub wyższa wskazuje na wystarczająco wysokie dopasowanie.

Fuzzy Matching (cont)

Warto pamiętać, że dostępność i składnia funkcji do fuzzy matching mogą się różnić w zależności od używanej bazy danych. Należy sprawdzić dokumentację konkretnej bazy danych w celu uzyskania informacji na temat dostępnych funkcji i ich zastosowania.

Operator NOT

Operator NOT jest używany w języku SQL do negacji logicznej warunku. Może być stosowany zarówno w warunkach logicznych, jak i w połączeniu z innymi operatorami.
Podstawowym zastosowaniem operatora NOT jest odwrócenie wartości logicznej wyrażenia. Jeśli wyrażenie logiczne jest prawdziwe, operator NOT zwróci wartość fałszu, a jeśli wyrażenie logiczne jest fałszywe, operator NOT zwróci wartość prawdy.
Przykład:

```
SELECT *
FROM tabela
WHERE NOT kolumna = 'wartosc';
```

W powyższym przykładzie, operator NOT jest używany do odwrócenia wyniku porównania wartości w kolumnie kolumna z określoną wartością. Jeśli wartość w kolumnie nie jest równa 'wartosc', warunek zostanie spełniony i wiersz zostanie zwrócony.
Operator NOT może być również używany do negacji warunków logicznych, takich jak AND i OR. Na przykład:

```
SELECT *
FROM tabela
WHERE NOT (warunek1 AND warunek2);
```

Operator NOT (cont)

W powyższym przykładzie, operator NOT jest używany do negacji całego warunku logicznego (warunek1 AND warunek2). Jeśli warunek ten jest prawdziwy, operator NOT zwróci wartość fałszu, co oznacza, że warunek zostanie spełniony i wiersz zostanie zwrócony.
Operator NOT może być również łączony z innymi operatorami logicznymi, takimi jak LIKE, BETWEEN, IN itp., aby tworzyć bardziej złożone wyrażenia warunkowe.
Warto pamiętać, że składnia i zachowanie operatora NOT mogą się różnić w zależności od konkretnej bazy danych. Należy zawsze sprawdzić dokumentację danej bazy danych w celu uzyskania dokładnej składni i zachowania operatora NOT.

IF ELSE

1. Konstrukcja `CASE WHEN`:

```
SELECT
   column _name,
    CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        ELSE result _de fault
    END AS result
FROM table_ name;
```

Przykład:

```
SELECT
    produc t_name,
    CASE
        WHEN units_sold > 1000 THEN 'High'
```

IF ELSE (cont)

```
        WHEN units_sold > 500 THEN 'Medium'
        ELSE 'Low'
    END AS sales_ cat egory
FROM products;
```

2. Funkcja `IF`:

```
SELECT
    column _name,
    IF(con dition, result _true, result _false) AS result
FROM table_ name;
```

Przykład:

```
SELECT
    produc t_name,
    IF(uni ts_sold > 1000, 'High', 'Low') AS sales_ cat egor
y
FROM products;
```

3. Konstrukcja `IIF` (dostępna w niektórych systemach baz danych):

```
SELECT
    column _name,
    IIF(co ndi tion, result _true, result _false) AS result
FROM table_ name;
```

Przykład:

```
SELECT
    produc t_name,
    IIF(un its _sold > 1000, 'High', 'Low') AS sales_ cat e
gory
FROM products;
```

Wszystkie te opcje pozwalają na definiowanie warunków i przypisywanie wartości w zależności od spełnienia tych warunków. Wybór odpowiedniej opcji zależy od systemu bazodanowego, który używasz, ponieważ nie wszystkie bazy danych obsługują wszystkie te konstrukcje. Należy sprawdzić dokumentację konkretnego systemu bazodanowego, aby dowiedzieć się, które opcje są dostępne i jak je prawidłowo używać.

co to jest Constraint?

Constraint (ograniczenie) w kontekście baz danych odnosi się do reguł i warunków, które są narzucane na dane w tabelach w celu zapewnienia integralności danych i utrzymania spójności bazy danych. Ograniczenia określają pewne reguły, którym muszą odpowiadać dane wprowadzane do tabeli. Główne typy ograniczeń w bazach danych to:

1. Primary Key (Klucz podstawowy): Ograniczenie primary key zapewnia unikalność wartości w kolumnie lub grupie kolumn w tabeli. Pozwala to na jednoznaczne identyfikowanie każdego wiersza w tabeli.

Przykład:

```
CREATE TABLE customers (
    custom er_id INT PRIMARY KEY,
    custom er_name VARCHA R(50)
);
```

2. Foreign Key (Klucz obcy): Ograniczenie foreign key definiuje relacje między tabelami. Określa, że wartości w kolumnie lub grupie kolumn w jednej tabeli muszą odpowiadać wartościom klucza podstawowego w innej tabeli.

Przykład:

co to jest Constraint? (cont)

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    custom er_id INT,
    order_date DATE,
    FOREIGN KEY (custo mer_id) REFERENCES customers (custo mer_i
d)
);
```

3. Unique Key (Klucz unikalny): Ograniczenie unique key zapewnia unikalność wartości w kolumnie lub grupie kolumn, ale pozwala na obecność wartości null.

Przykład:

```
CREATE TABLE employees (
    employ ee_id INT PRIMARY KEY,
    employ ee_name VARCHA R(50),
    email VARCHA R(50) UNIQUE
);
```

4. Check Constraint (Ograniczenie sprawdzające): Ograniczenie check definiuje warunek, który musi zostać spełniony przez wartości w kolumnie.

Przykład:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    produc t_name VARCHA R(50),
    quantity INT,
    price DECIMA L(10, 2),
    CHECK (quantity >= 0 AND price > 0)
```

co to jest Constraint? (cont)

```
);
```

Ograniczenia są ważnym elementem projektowania bazy danych i pomagają utrzymać spójność i integralność danych. Zapewniają również ochronę przed wprowadzaniem nieprawidłowych lub niepożądanych danych do tabel.