`SELECT` ifadesi SQL'de en temel ifadelerden biridir. Bu ifade, belirli sütunlardan veri seçmek için kullanılır ve genellikle veritabanından bilgi almak için kullanılır.

İşte birkaç `SELECT` örneği:

### Örnek Kullanımlar

1. \*\*Tüm Sütunları Seçmek:\*\*

```sql

SELECT \* FROM sales;

```

Bu sorgu, `sales` tablosundaki tüm sütunları ve tüm satırları seçer.

2. \*\*Belirli Sütunları Seçmek:\*\*

```sql

SELECT product\_No, product\_price, product\_delivery FROM sales;

```

Bu sorgu, `sales` tablosundaki `product\_No`, `product\_price` ve `product\_delivery` sütunlarını seçer.

3. \*\*Toplam Satır Sayısını Almak:\*\*

```sql

SELECT COUNT(\*) FROM sales;

```

Bu sorgu, `sales` tablosundaki toplam satır sayısını döndürür.

4. \*\*Belirli Koşulları Sağlayan Satırları Seçmek:\*\*

```sql

SELECT \* FROM sales WHERE product\_price > 500;

```

Bu sorgu, `sales` tablosundaki `product\_price` sütununda 500'den büyük olan tüm satırları seçer.

5. \*\*Toplama İşlemleri Yapmak:\*\*

```sql

SELECT SUM(product\_price) AS total\_sales FROM sales;

```

Bu sorgu, `sales` tablosundaki `product\_price` sütununun toplamını hesaplar ve sonucu `total\_sales` olarak adlandırır.

6. \*\*Tekrar Eden Satırları Kaldırmak:\*\*

```sql

SELECT DISTINCT product\_delivery FROM sales;

```

Bu sorgu, `sales` tablosundaki `product\_delivery` sütunundaki tekrar eden değerleri kaldırır ve benzersiz değerleri döndürür.

7. \*\*Sıralama Yapmak:\*\*

```sql

SELECT \* FROM sales ORDER BY product\_price DESC;

```

Bu sorgu, `sales` tablosundaki verileri `product\_price` sütununa göre azalan şekilde sıralar.

### Kullanımı

`SELECT` ifadesini, veritabanından belirli verileri almak için kullanabilirsiniz. `FROM` ifadesi ile hangi tablodan veri alınacağını belirtirsiniz. İsterseniz `WHERE` ifadesi ile belirli

koşulları sağlayan satırları seçebilir veya `GROUP BY` ifadesi ile veriyi gruplayabilirsiniz. Özel durumlarda, veri toplamak için `SUM()`, `COUNT()`, `AVG()` gibi toplama fonksiyonlarını kullanabilirsiniz.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

`GROUP BY` ifadesi SQL'de genellikle toplu işlemler için kullanılır. Bu ifade ile bir veya daha fazla sütuna göre gruplama yaparak, her gruba ait özet bilgiler alabiliriz. `GROUP BY` ile birlikte genellikle `SUM()`, `COUNT()`, `AVG()`, `MIN()`, `MAX()` gibi toplama fonksiyonları kullanılır.

Örneğin, `sales` tablosunda `product\_delivery` sütununa göre gruplama yaparak her teslimat türüne göre toplam ürün ağırlığını, ortalama fiyatı veya toplam ürün sayısını bulabilirsiniz.

### Örnek Kullanımlar

1. \*\*Teslimat Türüne Göre Toplam Ağırlık:\*\*

```sql

SELECT product\_delivery, SUM(product\_weight) AS total\_weight

FROM sales

GROUP BY product\_delivery;

GO

```

Bu sorgu, her teslimat türü için toplam ürün ağırlığını hesaplar.

2. \*\*Teslimat Türüne Göre Ortalama Fiyat:\*\*

```sql

SELECT product\_delivery, AVG(product\_price) AS average\_price

FROM sales

GROUP BY product\_delivery;

GO

```

Bu sorgu, her teslimat türü için ortalama ürün fiyatını hesaplar.

3. \*\*Teslimat Türüne Göre Ürün Sayısı:\*\*

```sql

SELECT product\_delivery, COUNT(\*) AS product\_count

FROM sales

GROUP BY product\_delivery;

GO

```

Bu sorgu, her teslimat türü için ürün sayısını hesaplar.

### Örnekler ile Birlikte

Şimdi yukarıdaki örneklerin tümünü içeren bir örnek yapalım:

```sql

USE sakontrolo2;

-- 'product\_delivery' sütununa göre gruplama ve toplama fonksiyonları ile örnek sorgular

SELECT product\_delivery,

COUNT(\*) AS product\_count,

SUM(product\_weight) AS total\_weight,

AVG(product\_price) AS average\_price,

MIN(product\_price) AS min\_price,

MAX(product\_price) AS max\_price

FROM sales

GROUP BY product\_delivery;

GO

```

Bu sorgu:

- Her teslimat türü için ürün sayısını (`COUNT(\*)`),

- Toplam ürün ağırlığını (`SUM(product\_weight)`),

- Ortalama ürün fiyatını (`AVG(product\_price)`),

- Minimum ürün fiyatını (`MIN(product\_price)`),

- Maksimum ürün fiyatını (`MAX(product\_price)`)

hesaplar ve sonuçları döndürür.

### Tabloyu Kontrol Etme

Bu sorguların doğru çalışması için `sales` tablosunun güncellenmiş ve doğru veriler içerdiğinden emin olalım:

```sql

-- Mevcut tabloyu kontrol et

SELECT \* FROM sales;

GO

```

Bu şekilde, mevcut `sales` tablosundaki verileri görüntüleyebilir ve güncelleme işlemlerinin doğru şekilde yapılıp yapılmadığını kontrol edebilirsiniz.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Evet, `SELECT` komutu SQL sorgularında oldukça esnek bir yapıya sahiptir ve birçok farklı ifadeyle birlikte kullanılabilir. İşte `SELECT` komutunda sıkça kullanılan bazı ifadeler:

1. \*\*`WHERE`:\*\* Belirli koşulları sağlayan satırları seçmek için kullanılır.

Örneğin:

```sql

SELECT \* FROM sales WHERE product\_price > 100;

```

2. \*\*`HAVING`:\*\* `GROUP BY` ile gruplandırılmış veriler üzerinde filtreleme yapmak için kullanılır.

Örneğin:

```sql

SELECT product\_delivery, COUNT(\*) AS delivery\_count

FROM sales

GROUP BY product\_delivery

HAVING COUNT(\*) > 10;

```

3. \*\*`DISTINCT`:\*\* Tekrar eden satırları kaldırmak için kullanılır.

Örneğin:

```sql

SELECT DISTINCT product\_delivery FROM sales;

```

4. \*\*Toplama Fonksiyonları (`SUM()`, `AVG()`, `COUNT()`, `MIN()`, `MAX()`):\*\* Belirli sütunların toplamını, ortalamasını, sayısını, minimum veya maksimum değerini hesaplamak için kullanılır.

Örneğin:

```sql

SELECT product\_delivery, COUNT(\*) AS delivery\_count

FROM sales

GROUP BY product\_delivery;

```

5. \*\*`JOIN`:\*\* İki veya daha fazla tabloyu birleştirmek için kullanılır.

Örneğin:

```sql

SELECT customers.customer\_name, orders.order\_date

FROM customers

INNER JOIN orders ON customers.customer\_id = orders.customer\_id;

```

6. \*\*`LIMIT` (veya `TOP`):\*\* Sonuç kümesinin belirli bir sayıda satırını döndürmek için kullanılır.

Örneğin:

```sql

SELECT \* FROM sales LIMIT 10;

```

7. \*\*Başka Alt Sorgular:\*\* `SELECT` ifadesi içinde başka `SELECT` ifadeleri kullanarak alt sorgular oluşturulabilir.

Örneğin:

```sql

SELECT product\_name

FROM products

WHERE product\_id IN (SELECT product\_id FROM sales WHERE sale\_date = '2022-01-01');

```

8. \*\*Başka Fonksiyonlar ve İşlemler:\*\* İşlem yapmak için matematiksel operatörler, dize işlevleri, tarih ve saat işlevleri vb. kullanılabilir.

Bu ifadeleri kullanarak, `SELECT` ifadesini oldukça güçlü bir şekilde özelleştirebilir ve istediğiniz sonuçları elde edebilirsiniz.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

İç birleştirme (inner join), iki veya daha fazla tabloyu birleştirmek için kullanılan bir SQL operasyonudur. İki tabloyu birleştirirken, bir tablodaki her satırı, diğer tablodaki ilgili satırlarla eşleştirir ve yalnızca eşleşen satırları içeren bir sonuç kümesi döndürür.

İşte iç birleştirme kullanarak iki tabloyu birleştirmek için temel bir SQL sözdizimi:

```sql

SELECT columns

FROM table1

INNER JOIN table2 ON table1.column = table2.column;

```

Bu sözdiziminde:

- `columns`: Seçmek istediğiniz sütunların listesi.

- `table1`, `table2`: Birleştirilmek istenen tabloların adları.

- `table1.column`, `table2.column`: Birleştirme için kullanılan sütunların adları.

Örnek bir iç birleştirme sorgusu:

Varsayalım ki bir kütüphane veritabanınız var ve `books` ve `authors` adında iki tablonuz var. Her kitap bir yazarın kimliğine (`author\_id`) referans verir. İşte bu durumu iç birleştirme ile birleştirme:

```sql

SELECT books.title, authors.name

FROM books

INNER JOIN authors ON books.author\_id = authors.author\_id;

```

Bu sorgu, her bir kitabın adını (`books.title`) ve yazarının adını (`authors.name`) birleştirir ve eşleşen her kitap ve yazar için bir satır döndürür.

İç birleştirme, bir tabloda eşleşen satırların olduğu durumlarda kullanılır. Eğer bir tabloda eşleşen satır yoksa, bu satır iç birleştirme sonucunda görünmez.

Bu, tablolar arasında eşleşen kayıtların olduğu durumlarda kullanışlıdır.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Sol birleştirme (left join) ve sağ birleştirme (right join), iç birleştirmenin (inner join) genişletilmiş versiyonlarıdır. Bu birleştirme türleri, tablolar arasında eşleşmeyen satırları da içeren sonuçlar oluşturur. Sol birleştirme, sol tablodaki tüm satırları ve sağ tablodaki eşleşen satırları birleştirirken, sağ birleştirme sağ tablodaki tüm satırları ve sol tablodaki eşleşen satırları birleştirir.

Sol birleştirme (LEFT JOIN) kullanımı:

```sql

SELECT columns

FROM table1

LEFT JOIN table2 ON table1.column = table2.column;

```

Sağ birleştirme (RIGHT JOIN) kullanımı:

```sql

SELECT columns

FROM table1

RIGHT JOIN table2 ON table1.column = table2.column;

```

Bu sözdiziminde:

- `columns`: Seçmek istediğiniz sütunların listesi.

- `table1`, `table2`: Birleştirilmek istenen tabloların adları.

- `table1.column`, `table2.column`: Birleştirme için kullanılan sütunların adları.

Sol birleştirme (LEFT JOIN) örneği:

```sql

SELECT books.title, authors.name

FROM books

LEFT JOIN authors ON books.author\_id = authors.author\_id;

```

Bu sorgu, her bir kitabın adını (`books.title`) ve yazarının adını (`authors.name`) birleştirir. Eğer bir kitap bir yazarın kimliğine (`author\_id`) sahip değilse, bu sorgu bu kitabı da döndürür.

Sağ birleştirme (RIGHT JOIN) örneği:

```sql

SELECT books.title, authors.name

FROM books

RIGHT JOIN authors ON books.author\_id = authors.author\_id;

```

Bu sorgu, her bir yazarın adını (`authors.name`) ve bu yazarın yazdığı kitapların adını (`books.title`) birleştirir. Eğer bir yazar hiç kitap yazmamışsa, bu sorgu bu yazarı da döndürür.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Evet, `HAVING` ve `WHERE` ifadeleri birlikte kullanılabilir ve genellikle farklı amaçlar için kullanılırlar. `WHERE` ifadesi, gruplama işlemi gerçekleşmeden önce satırları filtrelemek için kullanılırken, `HAVING` ifadesi ise gruplama işlemi gerçekleştikten sonra grupları filtrelemek için kullanılır.

`WHERE` ifadesi, `SELECT`, `INSERT`, `UPDATE`, ve `DELETE` sorgularında kullanılırken, `HAVING` ifadesi genellikle `GROUP BY` ifadesiyle birlikte kullanılır.

Aşağıda `WHERE` ve `HAVING` ifadelerinin birlikte nasıl kullanılacağını gösteren bir örnek bulunmaktadır.

### Örnek Senaryo:

Bir veritabanında `sales` adlı bir tablo olduğunu ve bu tabloda satış verilerinin tutulduğunu varsayalım. Her satırda ürün numarası (`product\_No`), satış miktarı (`product\_price`), ve teslimat şirketi numarası (`delivery\_company\_no`) bulunmaktadır.

### Örnek Veri:

```sql

CREATE TABLE sales (

product\_No VARCHAR(5),

product\_price DECIMAL(5, 2),

delivery\_company\_no VARCHAR(5)

);

INSERT INTO sales (product\_No, product\_price, delivery\_company\_no) VALUES

('00001', 123.45, '03001'),

('00002', 678.90, '03001'),

('00003', 234.56, '03002'),

('00004', 345.67, '03003'),

('00005', 456.78, '03003'),

('00006', 567.89, '03001'),

('00007', 678.90, '03002'),

('00008', 789.01, '03003'),

('00009', 890.12, '03002'),

('00010', 901.23, '03001');

```

### Sorgu:

Belirli bir teslimat şirketi için ortalama satış fiyatı 500'den fazla olan ürünleri listelemek istiyoruz. Bu durumda, `WHERE` ifadesi ile belirli bir teslimat şirketine ait kayıtları filtreleyebiliriz ve `HAVING` ifadesi ile gruplama sonrası ortalama satış fiyatını filtreleyebiliriz.

```sql

SELECT delivery\_company\_no, AVG(product\_price) AS avg\_price

FROM sales

WHERE delivery\_company\_no = '03001' -- İlk filtreleme: belirli bir teslimat şirketi için

GROUP BY delivery\_company\_no

HAVING AVG(product\_price) > 500; -- İkinci filtreleme: ortalama satış fiyatı 500'den fazla olanlar

```

### Açıklama:

1. `WHERE delivery\_company\_no = '03001'`: `sales` tablosundaki yalnızca `delivery\_company\_no` sütunu '03001' olan satırları filtreler.

2. `GROUP BY delivery\_company\_no`: Filtrelenmiş satırları `delivery\_company\_no` sütununa göre gruplar.

3. `HAVING AVG(product\_price) > 500`: Gruplanmış satırlar arasından, ortalama `product\_price` değeri 500'den fazla olan grupları seçer.

Bu şekilde, `WHERE` ve `HAVING` ifadeleri birlikte kullanılarak önce satırlar filtrelenir ve sonra gruplar üzerinde ek filtrelemeler yapılır.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

`UPDATE` komutu, SQL veritabanında bir tablodaki mevcut verileri değiştirmek için kullanılır. `UPDATE` komutunu kullanarak belirli koşullara uyan satırları değiştirebilirsiniz.

### `UPDATE` Komutunun Kullanımı

```sql

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

```

### Örnek

Önceki örnekleri kullanarak, `sales` tablosundaki belirli bir şirketin teslimatlarını güncellemek istiyorsanız:

#### Örnek 1: Belirli bir ürünü güncelleme

`product\_No` değeri '00001' olan satırın `product\_weight` ve `product\_netprice` değerlerini güncellemek:

```sql

UPDATE sales

SET product\_weight = 5, product\_netprice = 128.45

WHERE product\_No = '00001';

```

#### Örnek 2: Belirli bir şirkete ait tüm ürünleri güncelleme

Teslimat şirketi '03001' olan tüm satırların `product\_price` değerlerini %10 artırmak:

```sql

UPDATE sales

SET product\_price = product\_price \* 1.10

WHERE delivery\_company\_no = '03001';

```

#### Örnek 3: JOIN ile güncelleme

Teslimat şirketi 'glovo' olan tüm satırların `product\_weight` değerlerini 50 artırmak:

```sql

UPDATE s

SET s.product\_weight = s.product\_weight + 50

FROM sales s

INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no

WHERE d.company\_name = 'glovo';

```

### Güncellemeyi Gerçekleştirme

Önceki örneklerimizi kullanarak bir `UPDATE` işlemi gerçekleştirelim. Örneğin, `sales` tablosundaki belirli koşullara uyan satırları güncellemek isteyebiliriz:

#### Örnek Senaryo:

`delivery\_company\_no` '03001' olan satırların `product\_weight` değerini güncellemek.

### Adımlar:

1. \*\*Tabloların Yapısı ve Verileri:\*\*

- `sales` tablosu:

```sql

CREATE TABLE sales (

product\_No VARCHAR(5),

product\_price DECIMAL(5, 2),

product\_bill DECIMAL(5, 2),

product\_netprice DECIMAL(5, 2),

product\_weight FLOAT,

delivery\_company\_no VARCHAR(5),

CONSTRAINT product\_no\_l CHECK (LEN(product\_No) = 5),

CONSTRAINT company\_no\_l CHECK (LEN(delivery\_company\_no) = 5)

);

INSERT INTO sales (product\_No, product\_price, product\_bill, product\_netprice, product\_weight, delivery\_company\_no) VALUES

('00001', 123.45, 10, 133.45, 50, '03001'),

('00002', 678.90, 20, 698.90, 150, '03001'),

('00003', 234.56, 25, 259.56, 70, '03002'),

('00004', 345.67, 15, 360.67, 80, '03003'),

('00005', 456.78, 22, 478.78, 90, '03003'),

('00006', 567.89, 12, 579.89, 110, '03001'),

('00007', 678.90, 18, 696.90, 120, '03002'),

('00008', 789.01, 28, 817.01, 130, '03003'),

('00009', 890.12, 30, 920.12, 140, '03002'),

('00010', 901.23, 5, 906.23, 160, '03001');

CREATE TABLE delivery (

company\_no VARCHAR(5) PRIMARY KEY,

company\_name VARCHAR(MAX),

company\_vehicle VARCHAR(MAX)

);

INSERT INTO delivery (company\_no, company\_name, company\_vehicle) VALUES

('03001', 'glovo', 'motorcycle'),

('03002', 'amanatebi.ge', 'automobile'),

('03003', 'lorenzo.ge', 'truck');

```

2. \*\*Güncelleme Sorgusu:\*\*

`delivery\_company\_no` '03001' olan satırların `product\_weight` değerini 10 artırmak:

```sql

UPDATE sales

SET product\_weight = product\_weight + 10

WHERE delivery\_company\_no = '03001';

```

Bu sorgu, `delivery\_company\_no` değeri '03001' olan tüm satırların `product\_weight` değerini 10 artırır.

### JOIN ile Güncelleme

Teslimat şirketi 'glovo' olan tüm satırların `product\_weight` değerlerini 50 artırmak:

```sql

UPDATE s

SET s.product\_weight = s.product\_weight + 50

FROM sales s

INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no

WHERE d.company\_name = 'glovo';

```

Bu sorgu, `delivery` tablosunda `company\_name` değeri 'glovo' olan satırlarla eşleşen `sales` tablosundaki satırların `product\_weight` değerini 50 artırır.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

`DELETE` komutu, SQL veritabanında bir tablodan belirli koşullara uyan satırları silmek için kullanılır. Bu komut genellikle WHERE koşulu ile birlikte kullanılır. Eğer WHERE koşulu belirtilmezse, tablodaki tüm satırlar silinir.

### `DELETE` Komutunun Kullanımı

```sql

DELETE FROM table\_name WHERE condition;

```

### Örnekler

#### Örnek 1: Belirli bir ürünü silme

`product\_No` değeri '00001' olan satırı silmek:

```sql

DELETE FROM sales WHERE product\_No = '00001';

```

#### Örnek 2: Belirli bir şirkete ait tüm ürünleri silme

`delivery\_company\_no` değeri '03001' olan tüm satırları silmek:

```sql

DELETE FROM sales WHERE delivery\_company\_no = '03001';

```

### JOIN ile Silme

Birden fazla tabloyu birleştirerek belirli koşullara uyan satırları silmek için JOIN kullanabilirsiniz.

#### Örnek: JOIN kullanarak silme

Teslimat şirketi 'glovo' olan tüm satırları silmek:

```sql

DELETE s

FROM sales s

INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no

WHERE d.company\_name = 'glovo';

```

### Adımlar:

1. \*\*Tabloların Yapısı ve Verileri:\*\*

- `sales` tablosu:

```sql

CREATE TABLE sales (

product\_No VARCHAR(5),

product\_price DECIMAL(5, 2),

product\_bill DECIMAL(5, 2),

product\_netprice DECIMAL(5, 2),

product\_weight FLOAT,

delivery\_company\_no VARCHAR(5),

CONSTRAINT product\_no\_l CHECK (LEN(product\_No) = 5),

CONSTRAINT company\_no\_l CHECK (LEN(delivery\_company\_no) = 5)

);

INSERT INTO sales (product\_No, product\_price, product\_bill, product\_netprice, product\_weight, delivery\_company\_no) VALUES

('00001', 123.45, 10, 133.45, 50, '03001'),

('00002', 678.90, 20, 698.90, 150, '03001'),

('00003', 234.56, 25, 259.56, 70, '03002'),

('00004', 345.67, 15, 360.67, 80, '03003'),

('00005', 456.78, 22, 478.78, 90, '03003'),

('00006', 567.89, 12, 579.89, 110, '03001'),

('00007', 678.90, 18, 696.90, 120, '03002'),

('00008', 789.01, 28, 817.01, 130, '03003'),

('00009', 890.12, 30, 920.12, 140, '03002'),

('00010', 901.23, 5, 906.23, 160, '03001');

CREATE TABLE delivery (

company\_no VARCHAR(5) PRIMARY KEY,

company\_name VARCHAR(MAX),

company\_vehicle VARCHAR(MAX)

);

INSERT INTO delivery (company\_no, company\_name, company\_vehicle) VALUES

('03001', 'glovo', 'motorcycle'),

('03002', 'amanatebi.ge', 'automobile'),

('03003', 'lorenzo.ge', 'truck');

```

2. \*\*Silme Sorguları:\*\*

#### Örnek 1: `product\_No` değeri '00001' olan satırı silmek

```sql

DELETE FROM sales WHERE product\_No = '00001';

```

#### Örnek 2: `delivery\_company\_no` değeri '03001' olan tüm satırları silmek

```sql

DELETE FROM sales WHERE delivery\_company\_no = '03001';

```

#### Örnek 3: JOIN kullanarak `company\_name` değeri 'glovo' olan teslimat şirketine ait satırları silmek

```sql

DELETE s

FROM sales s

INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no

WHERE d.company\_name = 'glovo';

```

### Açıklama:

- \*\*`DELETE FROM sales WHERE product\_No = '00001';`\*\*: `product\_No` değeri '00001' olan satırı siler.

- \*\*`DELETE FROM sales WHERE delivery\_company\_no = '03001';`\*\*: `delivery\_company\_no` değeri '03001' olan tüm satırları siler.

- \*\*`DELETE s FROM sales s INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no WHERE d.company\_name = 'glovo';`\*\*: Teslimat şirketi 'glovo' olan tüm satırları siler.

Bu sorgular, SQL veritabanınızda belirli koşullara uyan satırları silmek için kullanılır. Bu tür işlemleri dikkatli bir şekilde kullanmalısınız çünkü `DELETE` komutu geri alınamaz.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Stored procedures, SQL Server gibi ilişkisel veritabanı yönetim sistemlerinde tekrar kullanılabilir SQL kodlarını saklamak ve çalıştırmak için kullanılır. Stored procedure'lar, belirli bir görev veya işlem için bir dizi SQL ifadeleri içerir ve bir kere oluşturulduktan sonra birçok kez çağrılabilir.

### Stored Procedure Oluşturma ve Kullanma

1. \*\*Stored Procedure Oluşturma\*\*

```sql

CREATE PROCEDURE procedure\_name

AS

BEGIN

-- SQL ifadeleri buraya yazılır

END;

```

2. \*\*Stored Procedure Çağırma\*\*

```sql

EXEC procedure\_name;

```

### Örnekler

#### Örnek 1: Basit Bir Stored Procedure Oluşturma

Veritabanındaki tüm satışları listeleyen bir stored procedure oluşturma:

```sql

CREATE PROCEDURE GetAllSales

AS

BEGIN

SELECT \* FROM sales;

END;

```

Bu stored procedure'ü çağırmak için:

```sql

EXEC GetAllSales;

```

#### Örnek 2: Parametre Alan Stored Procedure Oluşturma

Belirli bir fiyatın üzerindeki satışları listeleyen bir stored procedure oluşturma:

```sql

CREATE PROCEDURE GetSalesAbovePrice

@Price DECIMAL(5, 2)

AS

BEGIN

SELECT \* FROM sales

WHERE product\_price > @Price;

END;

```

Bu stored procedure'ü çağırmak için:

```sql

EXEC GetSalesAbovePrice @Price = 500.00;

```

#### Örnek 3: Güncelleme Yapan Stored Procedure

Belirli bir ürünü güncelleyen bir stored procedure oluşturma:

```sql

CREATE PROCEDURE UpdateProductPrice

@ProductNo VARCHAR(5),

@NewPrice DECIMAL(5, 2)

AS

BEGIN

UPDATE sales

SET product\_price = @NewPrice

WHERE product\_No = @ProductNo;

END;

```

Bu stored procedure'ü çağırmak için:

```sql

EXEC UpdateProductPrice @ProductNo = '00001', @NewPrice = 150.00;

```

#### Örnek 4: JOIN Kullanarak Veri Döndüren Stored Procedure

Teslimat şirketleri ile satışları birleştiren ve belirli koşullara göre filtreleyen bir stored procedure oluşturma:

```sql

CREATE PROCEDURE GetSalesWithDelivery

@MaxPrice DECIMAL(5, 2)

AS

BEGIN

SELECT s.product\_No, s.product\_price, s.product\_bill, s.product\_netprice, s.product\_weight, d.company\_name

FROM sales s

INNER JOIN delivery d ON s.delivery\_company\_no = d.company\_no

WHERE s.product\_price <= @MaxPrice;

END;

```

Bu stored procedure'ü çağırmak için:

```sql

EXEC GetSalesWithDelivery @MaxPrice = 500.00;

```

### Özet:

- \*\*Stored procedures\*\*, belirli bir görev için bir grup SQL ifadesini içerir.

- Tekrar kullanılabilir, parametrik olabilir ve çeşitli işlemler gerçekleştirebilir.

- Kullanımı basit olup veritabanı yönetimini kolaylaştırır.

Stored procedures, özellikle büyük ve karmaşık veritabanı işlemlerinde performans ve yönetim kolaylığı sağlar. Bu nedenle, veritabanı uygulamalarında yaygın olarak kullanılır.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### TRY...CATCH Bloklarının Detaylı Açıklaması

TRY...CATCH blokları, SQL Server'da hata yönetimi için kullanılan yapılar olup, bir hata meydana geldiğinde kodun güvenli bir şekilde nasıl ele alınacağını sağlar. İşlemlerin güvenli ve tutarlı bir şekilde yürütülmesini garanti etmek için özellikle transaction'lar ile birlikte sıkça kullanılır.

### Temel Yapısı

```sql

BEGIN TRY

-- Hata meydana gelebilecek SQL komutları

END TRY

BEGIN CATCH

-- Hata meydana geldiğinde yürütülecek SQL komutları

END CATCH

```

### TRY...CATCH Kullanım Senaryoları

1. \*\*Basit Hata Yakalama\*\*

- TRY bloğu içinde hataya neden olabilecek SQL komutları bulunur.

- CATCH bloğu, hatanın meydana geldiği durumlarda çalışarak hata yönetimi sağlar.

```sql

BEGIN TRY

-- Hata oluşturabilecek SQL komutu

SELECT 1 / 0;

END TRY

BEGIN CATCH

-- Hata mesajını döndür

PRINT 'Hata meydana geldi: ' + ERROR\_MESSAGE();

END CATCH;

```

2. \*\*Transaction Yönetimi\*\*

- Bir transaction başlatılır, TRY bloğunda işlem yapılır ve eğer başarılı olursa COMMIT edilir.

- Hata meydana gelirse CATCH bloğunda ROLLBACK yapılır.

```sql

BEGIN TRANSACTION;

BEGIN TRY

-- İşlemler

UPDATE sales SET product\_price = product\_price \* 1.1 WHERE product\_price < 100;

-- Transaction başarılı ise commit edilir

COMMIT;

END TRY

BEGIN CATCH

-- Hata durumunda rollback yapılır

ROLLBACK;

-- Hata mesajını döndür

PRINT 'Hata meydana geldi: ' + ERROR\_MESSAGE();

END CATCH;

```

3. \*\*Detaylı Hata Bilgileri Yakalama\*\*

- Hatanın detaylarını yakalayıp işlemi loglamak veya daha fazla bilgi sağlamak için kullanılır.

```sql

BEGIN TRY

-- Hata oluşturabilecek SQL komutu

EXEC sp\_executesql N'SELECT 1 / 0';

END TRY

BEGIN CATCH

DECLARE

@ErrorMessage NVARCHAR(4000),

@ErrorSeverity INT,

@ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Hata detaylarını loglamak veya mesaj döndürmek

PRINT 'Hata Mesajı: ' + @ErrorMessage;

PRINT 'Hata Şiddeti: ' + CAST(@ErrorSeverity AS NVARCHAR);

PRINT 'Hata Durumu: ' + CAST(@ErrorState AS NVARCHAR);

END CATCH;

```

### TRY...CATCH ve Transaction Kullanımı

Aşağıda, TRY...CATCH ve transaction'ları birleştirerek bir stored procedure içinde nasıl kullanabileceğinizi gösteren detaylı bir örnek bulunmaktadır:

```sql

-- Eğer 'UpdateSalesData' adında bir stored procedure varsa sil

IF EXISTS (SELECT \* FROM sys.objects WHERE type = 'P' AND name = 'UpdateSalesData')

BEGIN

DROP PROCEDURE UpdateSalesData;

END;

GO

-- Yeni bir stored procedure oluştur

CREATE PROCEDURE UpdateSalesData

AS

BEGIN

-- Transaction başlat

BEGIN TRANSACTION;

BEGIN TRY

-- 'sales' tablosunu güncelle

UPDATE sales

SET

product\_bill = CASE

WHEN product\_price BETWEEN 0 AND 100 THEN product\_price \* 0.02

ELSE product\_price \* 0.05

END,

product\_netprice = product\_price + CASE

WHEN product\_price BETWEEN 0 AND 100 THEN product\_price \* 0.02

ELSE product\_price \* 0.05

END;

-- Geçici tablo oluşturarak her satır için farklı rastgele değer atayalım

WITH RandomWeights AS (

SELECT product\_No, ROUND(1 + (RAND(CHECKSUM(NEWID())) \* (2000 - 1)), 2) AS random\_weight

FROM sales

)

UPDATE sales

SET product\_weight = RW.random\_weight

FROM sales S

JOIN RandomWeights RW ON S.product\_No = RW.product\_No;

-- 'product\_weight' sütununu belirli bir değerle güncelle

UPDATE sales

SET product\_weight = 5

WHERE product\_No = '00001';

-- 'delivery\_company\_no' sütununu güncelle

UPDATE sales

SET delivery\_company\_no = CASE

WHEN product\_weight BETWEEN 1 AND 10 THEN '03001'

WHEN product\_weight > 10 AND product\_weight < 150 THEN '03002'

ELSE '03003'

END;

-- Transaction'ı başarılı bir şekilde tamamla

COMMIT;

END TRY

BEGIN CATCH

-- Hata durumunda transaction'ı geri al

ROLLBACK;

-- Hata bilgilerini döndür

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

-- Hata mesajını döndür

RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

END;

GO

-- Stored procedure'ü çağır

EXEC UpdateSalesData;

GO

```

### Açıklamalar

1. \*\*BEGIN TRANSACTION\*\*: Yeni bir transaction başlatır.

2. \*\*BEGIN TRY\*\*: İşlem bloğunu başlatır ve buradaki SQL ifadelerini dener.

3. \*\*UPDATE\*\*: Belirtilen güncelleme işlemlerini gerçekleştirir.

4. \*\*COMMIT\*\*: Tüm işlemler başarılı olursa transaction'ı tamamlar.

5. \*\*BEGIN CATCH\*\*: Hata durumunda çalışacak blok.

6. \*\*ROLLBACK\*\*: Hata durumunda transaction'ı geri alır.

7. \*\*DECLARE\*\*: Hata mesajı, şiddeti ve durumu için değişkenleri tanımlar.

8. \*\*SELECT\*\*: Hata mesajı, şiddeti ve durumu bilgilerini değişkenlere atar.

9. \*\*RAISERROR\*\*: Hata mesajını döndürür.

Bu şekilde TRY...CATCH blokları, transaction'ların güvenli bir şekilde yönetilmesini ve hata durumunda işlemlerin geri alınmasını sağlar.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*\*Transaction (işlem)\*\*, bir veritabanı yönetim sisteminde bir veya daha fazla SQL komutunun bir arada atomik olarak yürütülmesini sağlayan bir mekanizmadır. Bu, transaction içindeki tüm işlemlerin başarılı bir şekilde tamamlanması veya tamamlanmaması durumunda hepsinin birden gerçekleştirilmesi veya hiç gerçekleştirilmemesi anlamına gelir.

### Transaction Özellikleri (ACID)

Transaction'lar dört temel özellik (ACID) sağlar:

1. \*\*Atomicity (Bütünlük)\*\*: Transaction içindeki tüm işlemler ya hep birlikte gerçekleştirilir ya da hiçbiri gerçekleştirilmez.

2. \*\*Consistency (Tutarlılık)\*\*: Transaction tamamlandığında, veritabanı tutarlı bir durumda olur.

3. \*\*Isolation (İzolasyon)\*\*: Bir transaction'ın ara durumları diğer transaction'lardan izole edilir.

4. \*\*Durability (Süreklilik)\*\*: Bir transaction tamamlandığında, sonuçlar kalıcı olarak saklanır.

### Transaction Komutları

- \*\*BEGIN TRANSACTION\*\*: Yeni bir transaction başlatır.

- \*\*COMMIT\*\*: Transaction'ı başarılı bir şekilde tamamlar ve yapılan değişiklikleri kalıcı hale getirir.

- \*\*ROLLBACK\*\*: Transaction'ı geri alır ve yapılan tüm değişiklikleri iptal eder.

### Transaction Örneği

Aşağıdaki örnekte, verilen güncelleme işlemlerini bir transaction içinde gerçekleştiren bir stored procedure oluşturuyoruz.

```sql

-- Eğer 'UpdateSalesData' adında bir stored procedure varsa sil

IF EXISTS (SELECT \* FROM sys.objects WHERE type = 'P' AND name = 'UpdateSalesData')

BEGIN

DROP PROCEDURE UpdateSalesData;

END;

GO

-- Yeni bir stored procedure oluştur

CREATE PROCEDURE UpdateSalesData

AS

BEGIN

-- Transaction başlat

BEGIN TRANSACTION;

BEGIN TRY

-- 'sales' tablosunu güncelle

UPDATE sales

SET

product\_bill = CASE

WHEN product\_price BETWEEN 0 AND 100 THEN product\_price \* 0.02

ELSE product\_price \* 0.05

END,

product\_netprice = product\_price + CASE

WHEN product\_price BETWEEN 0 AND 100 THEN product\_price \* 0.02

ELSE product\_price \* 0.05

END;

-- Geçici tablo oluşturarak her satır için farklı rastgele değer atayalım

WITH RandomWeights AS (

SELECT product\_No, ROUND(1 + (RAND(CHECKSUM(NEWID())) \* (2000 - 1)), 2) AS random\_weight

FROM sales

)

UPDATE sales

SET product\_weight = RW.random\_weight

FROM sales S

JOIN RandomWeights RW ON S.product\_No = RW.product\_No;

-- 'product\_weight' sütununu belirli bir değerle güncelle

UPDATE sales

SET product\_weight = 5

WHERE product\_No = '00001';

-- 'delivery\_company\_no' sütununu güncelle

UPDATE sales

SET delivery\_company\_no = CASE

WHEN product\_weight BETWEEN 1 AND 10 THEN '03001'

WHEN product\_weight > 10 AND product\_weight < 150 THEN '03002'

ELSE '03003'

END;

-- Transaction'ı başarılı bir şekilde tamamla

COMMIT;

END TRY

BEGIN CATCH

-- Hata durumunda transaction'ı geri al

ROLLBACK;

-- Hata mesajını döndür

THROW;

END CATCH;

END;

GO

-- Stored procedure'ü çağır

EXEC UpdateSalesData;

GO

```

### Açıklamalar

1. \*\*BEGIN TRANSACTION\*\*: Yeni bir transaction başlatır.

2. \*\*BEGIN TRY\*\*: Denenecek işlem bloğunu başlatır.

3. \*\*UPDATE\*\*: Belirtilen güncelleme işlemlerini gerçekleştirir.

4. \*\*COMMIT\*\*: Tüm işlemler başarılı olursa transaction'ı tamamlar.

5. \*\*BEGIN CATCH\*\*: Hata durumunda çalışacak blok.

6. \*\*ROLLBACK\*\*: Hata durumunda transaction'ı geri alır.

7. \*\*THROW\*\*: Hata mesajını döndürür.

Bu şekilde, stored procedure içindeki tüm işlemler bir transaction olarak yürütülür ve herhangi bir hata durumunda tüm işlemler geri alınır.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Elbette! SQL tetikleyicileri (triggers), veritabanında belirli olaylar gerçekleştiğinde otomatik olarak çalışan özel türde saklı prosedürlerdir. İş kurallarını uygulamak, veri doğrulaması yapmak ve izleme kayıtlarını tutmak gibi çeşitli görevler için kullanılırlar. Tetikleyiciler, bir tabloda ekleme (INSERT), güncelleme (UPDATE) ve silme (DELETE) işlemlerinden önce veya sonra çalışacak şekilde ayarlanabilir.

### Tetikleyici Türleri

1. \*\*BEFORE Tetikleyicileri\*\*: Tetikleyici olay (INSERT, UPDATE, DELETE) gerçekleşmeden önce çalıştırılır. Genellikle veri doğrulaması gibi işlemler için kullanılır.

2. \*\*AFTER Tetikleyicileri\*\*: Tetikleyici olay gerçekleşip tamamlandıktan sonra çalıştırılır. Veri güncellemelerinden sonra loglama veya başka tabloları güncelleme gibi işlemler için kullanılır.

### Örnekler

#### 1. BEFORE INSERT Tetikleyicisi

Bu örnek, bir `employees` tablosuna veri eklenmeden önce çalışacak bir tetikleyiciyi gösterir. Çalışanların maaşlarının belirli bir değerden düşük olmamasını sağlar.

```sql

CREATE TRIGGER before\_employee\_insert

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

IF NEW.salary < 3000 THEN

SET NEW.salary = 3000;

END IF;

END;

```

#### 2. AFTER INSERT Tetikleyicisi

Bu tetikleyici, yeni bir çalışan eklenmesinden sonra bir `audit\_log` tablosuna kayıt ekler.

```sql

CREATE TRIGGER after\_employee\_insert

AFTER INSERT ON employees

FOR EACH ROW

BEGIN

INSERT INTO audit\_log (employee\_id, action, action\_time)

VALUES (NEW.employee\_id, 'INSERT', NOW());

END;

```

#### 3. BEFORE UPDATE Tetikleyicisi

Bu tetikleyici, bir çalışanın maaşı güncellenmeden önce çalışır ve yeni maaşın eski maaştan çok fazla artmasını engeller.

```sql

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON employees

FOR EACH ROW

BEGIN

IF NEW.salary > OLD.salary \* 1.1 THEN

SET NEW.salary = OLD.salary \* 1.1;

END IF;

END;

```

### Tetikleyici Kullanım Alanları

- \*\*Veri Doğrulama\*\*: Giriş verilerinin belirli kurallara uygun olmasını sağlar.

- \*\*İş Kuralları\*\*: Örneğin, stok miktarı azaldığında otomatik sipariş oluşturma gibi iş süreçlerini otomatikleştirir.

- \*\*Denetim ve İzleme\*\*: Tablo değişikliklerini izleyip log tablosuna kayıt ekler.

- \*\*Veri Bütünlüğü\*\*: Bir tabloda yapılan değişikliklerin diğer tablolarla tutarlı olmasını sağlar.

SQL tetikleyicileri, veritabanı yönetiminde güçlü bir araçtır ve doğru kullanıldığında veri bütünlüğünü ve güvenliğini artırmada önemli bir rol oynar.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

SQL Server'da `IF...ELSE IF...ELSE` yapısını kullanarak, birden fazla koşulu değerlendirebilir ve bu koşullara göre farklı işlemler gerçekleştirebilirsiniz. İşte SQL Server'da `IF...ELSE IF...ELSE` yapısının nasıl kullanılacağını gösteren bir örnek:

### `IF...ELSE IF...ELSE` Yapısı

```sql

IF <koşul1>

BEGIN

-- Koşul1 doğruysa yapılacak işlemler

END

ELSE IF <koşul2>

BEGIN

-- Koşul2 doğruysa yapılacak işlemler

END

ELSE

BEGIN

-- Yukarıdaki koşullar yanlışsa yapılacak işlemler

END

```

### Örnek Prosedür: `IF...ELSE IF...ELSE` Kullanımı

Aşağıda `IF...ELSE IF...ELSE` yapısını içeren bir prosedür örneği verilmiştir. Bu prosedürde `pr\_quantity` değeri pozitif, sıfır veya negatif olup olmadığını kontrol eder ve ona göre farklı işlemler gerçekleştirir.

```sql

-- Eğer 'add\_product\_to\_shop' prosedürü varsa, sil

IF EXISTS (SELECT \* FROM sys.objects WHERE type = 'P' AND name = 'add\_product\_to\_shop')

DROP PROCEDURE add\_product\_to\_shop;

GO

-- 'add\_product\_to\_shop' prosedürünü oluştur

CREATE PROCEDURE add\_product\_to\_shop

@pr\_id varchar(10),

@shop\_id varchar(5),

@pr\_quantity int

AS

BEGIN

BEGIN TRY

-- Koşul1: pr\_quantity > 0

IF @pr\_quantity > 0

BEGIN

-- Eğer pr\_quantity > 0 ise, product\_shop tablosuna veri ekle

INSERT INTO product\_shop (product\_no, product\_count)

VALUES (@pr\_id, @pr\_quantity);

PRINT 'Product added successfully';

END

-- Koşul2: pr\_quantity = 0

ELSE IF @pr\_quantity = 0

BEGIN

-- Eğer pr\_quantity = 0 ise, hata mesajı göster

PRINT 'Quantity must be greater than 0';

END

-- Diğer durumlar: pr\_quantity < 0

ELSE

BEGIN

-- Eğer pr\_quantity < 0 ise, hata mesajı göster

PRINT 'Quantity cannot be negative';

END

END TRY

BEGIN CATCH

-- Hata durumunda bilgi al

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

DECLARE @ErrorLine INT;

SELECT

@ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE(),

@ErrorLine = ERROR\_LINE();

-- Hata mesajının gösterilmesi

PRINT 'Error occurred: ' + @ErrorMessage;

PRINT 'Line number: ' + CAST(@ErrorLine AS NVARCHAR(10));

PRINT 'Message: "Employee or department ID is invalid"';

END CATCH;

END;

GO

```

### Örnek Kullanım

Prosedürü çağırarak veri ekleyebilirsiniz:

```sql

-- Pozitif miktar

EXEC add\_product\_to\_shop @pr\_id = '0123456789', @shop\_id = '12345', @pr\_quantity = 20;

-- Sıfır miktar

EXEC add\_product\_to\_shop @pr\_id = '0123456799', @shop\_id = '12345', @pr\_quantity = 0;

-- Negatif miktar

EXEC add\_product\_to\_shop @pr\_id = '0523456789', @shop\_id = '12345', @pr\_quantity = -5;

```

Bu örnekte `pr\_quantity` değeri pozitifse ürün eklenir, sıfırsa "Quantity must be greater than 0" mesajı gösterilir ve negatifse "Quantity cannot be negative" mesajı gösterilir. Eğer prosedür çalıştırılmasında hata oluşursa, hata mesajı ve hata satırı numarası gösterilir.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

SQL Server'da fonksiyonlar (functions), belirli bir mantıksal işlemi gerçekleştiren ve genellikle veri döndüren nesnelerdir. SQL fonksiyonları genellikle veritabanı uygulamalarında tekrar kullanılabilirlik ve kodun yeniden kullanılabilirliği açısından önemlidir. SQL Server'da iki temel fonksiyon türü bulunur:

1. \*\*Scalar Fonksiyonlar (Scalar Functions)\*\*:

- Tek bir değer döndüren fonksiyonlardır.

- Genellikle SELECT ifadesinde bir sütun değeri veya WHERE koşullarında kullanılabilir.

- Örneğin, bir sayının karesini hesaplamak gibi basit matematiksel işlemler için kullanılabilir.

2. \*\*Table-valued Fonksiyonlar (Table-valued Functions)\*\*:

- Bir veya daha fazla satırı veya tabloyu döndüren fonksiyonlardır.

- Tek satır veya çoklu satırlar döndürebilirler.

- Örneğin, bir kategoriye ait tüm ürünleri listeleyen veya belirli bir tarihte yapılan tüm siparişleri döndüren fonksiyonlar gibi kullanılabilir.

### Scalar Fonksiyonlar

Scalar fonksiyonlar genellikle aşağıdaki gibi tanımlanır:

```sql

CREATE FUNCTION functionName

(

@param1 datatype,

@param2 datatype

)

RETURNS returnDatatype

AS

BEGIN

-- Fonksiyonun mantığı burada tanımlanır

DECLARE @result returnDatatype;

-- Örnek bir işlem: iki sayının toplamını döndürme

SET @result = @param1 + @param2;

-- Sonuç döndürülür

RETURN @result;

END;

```

Örnek bir scalar fonksiyonun tanımlanması ve kullanılması:

```sql

-- Scalar fonksiyon örneği: iki sayının toplamını döndüren fonksiyon

CREATE FUNCTION AddNumbers

(

@num1 INT,

@num2 INT

)

RETURNS INT

AS

BEGIN

DECLARE @sum INT;

SET @sum = @num1 + @num2;

RETURN @sum;

END;

GO

-- Fonksiyonu çağırma

SELECT dbo.AddNumbers(5, 3) AS SumResult; -- Output: 8

```

### Table-valued Fonksiyonlar

Table-valued fonksiyonlar genellikle aşağıdaki gibi tanımlanır:

```sql

CREATE FUNCTION functionName

(

@param1 datatype,

@param2 datatype

)

RETURNS TABLE

AS

RETURN

(

-- Fonksiyonun mantığı burada tanımlanır

SELECT column1, column2, ...

FROM tableName

WHERE condition

);

```

Örnek bir table-valued fonksiyonun tanımlanması ve kullanılması:

```sql

-- Table-valued fonksiyon örneği: belirli bir kategoriye ait ürünleri listeleyen fonksiyon

CREATE FUNCTION GetProductsByCategory

(

@categoryID INT

)

RETURNS TABLE

AS

RETURN

(

SELECT ProductID, ProductName, UnitPrice

FROM Products

WHERE CategoryID = @categoryID

);

GO

-- Fonksiyonu çağırma

SELECT \* FROM dbo.GetProductsByCategory(1); -- CategoryID = 1 olan ürünler listelenecek

```

### Fonksiyonların Önemi ve Kullanımı

SQL Server'da fonksiyonlar, kod tekrarını önler, karmaşık işlemleri modüler hale getirir ve sorguların veya prosedürlerin daha okunabilir ve bakımı kolay hale gelmesini sağlar. Veri manipülasyonu, iş kuralları ve veri dönüşümleri gibi birçok senaryoda kullanılabilirler. Ancak, fonksiyonların performans etkilerini dikkate almak önemlidir; bazı durumlarda fonksiyon kullanımı sorgu performansını olumsuz etkileyebilir.

SQL Server'da fonksiyonların detaylı kullanımı ve performans optimizasyonları için veri modelinize ve iş gereksinimlerinize uygun şekilde değerlendirme yapmanız önemlidir.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

SQL Server'da "views" (görünümler), bir veya birden fazla tablodan veri sağlayan, adlandırılmış ve önceden tanımlanmış sorguları içeren sanal tablolardır. Görünümler, veritabanı üzerinde sıkça kullanılan sorguları kolayca erişilebilir hale getirir ve tekrar kullanılabilirliği artırır. Ayrıca, veritabanı yapısını gizleyerek kullanıcıların verilere daha kolay erişmesini sağlar.

### Görünüm (View) Oluşturma

Görünüm oluşturmak için `CREATE VIEW` komutu kullanılır. Örneğin, `depo` tablosundan belirli sütunları içeren bir görünüm oluşturalım:

```sql

-- Eğer 'vw\_depo\_summary' adında bir görünüm varsa, önce silelim

IF EXISTS (SELECT \* FROM sys.views WHERE name = 'vw\_depo\_summary')

DROP VIEW vw\_depo\_summary;

GO

-- 'depo' tablosundan bir görünüm oluşturalım

CREATE VIEW vw\_depo\_summary

AS

SELECT pr\_id, pr\_name, unit\_price

FROM depo;

GO

```

Yukarıdaki örnekte, `vw\_depo\_summary` adında bir görünüm oluşturduk. Bu görünüm, `depo` tablosundan `pr\_id`, `pr\_name` ve `unit\_price` sütunlarını içerir. Görünüm oluşturulduktan sonra, `vw\_depo\_summary` adıyla bu yapıyı kullanarak `SELECT` sorguları yapabilirsiniz.

### Görünüm Kullanımı

Oluşturduğumuz `vw\_depo\_summary` görünümünü kullanarak bir `SELECT` sorgusu yapalım:

```sql

-- Oluşturduğumuz görünümü kullanarak sorgu yapalım

SELECT \*

FROM vw\_depo\_summary;

```

Bu sorgu, `vw\_depo\_summary` görünümünden tüm satırları ve sütunları seçer.

### Görünüm Avantajları

1. \*\*Kod Tekrarını Azaltma\*\*: Sık kullanılan sorguları tek bir noktada tanımlayarak, kod tekrarını önler.

2. \*\*Veri Erişimini Kolaylaştırma\*\*: Kullanıcıların karmaşık sorguları bilmeden ve doğrudan erişim sağlamalarını sağlar.

3. \*\*Veri Güvenliği ve Kullanıcı İzolasyonu\*\*: Görünümler, kullanıcıların doğrudan tablolara erişimini kısıtlayarak veri güvenliğini artırır.

4. \*\*Performans İyileştirmesi\*\*: Veritabanı optimizasyonu için optimize edilmiş görünümler kullanılabilir.

Görünümler, veritabanı uygulamalarında veri erişimini yönetmek ve kod karmaşıklığını azaltmak için güçlü bir araçtır. Her bir görünüm, belirli bir işlevi veya perspektifi temsil eder ve veri tabanı tasarımını daha modüler ve yönetilebilir hale getirir.