



DBS

UDF / SP / Erweitertes SQL

Prof. Dr. Patrick Cato

User-Defined Functions (UDF)

These: „Keep the data on the server!“

- **Leistungssteigerung:** Da sie auf dem Datenbankserver ausgeführt werden, können sie Netzwerklatenzen reduzieren, indem sie mehrere SQL-Anweisungen in einer einzigen Datenbankabfrage bündeln.
- **Modularität:** Sie ermöglichen es, häufig verwendete oder komplexe Funktionen in einer zentralen Stelle zu speichern und von verschiedenen Anwendungen oder Benutzern aufzurufen.
- **Sicherheit:** Sie können den Zugriff auf bestimmte Datenbankoperationen kontrollieren, indem sie Berechtigungen auf Stored Procedures statt auf Tabellen oder Ansichten setzen.
- **Wartbarkeit:** Änderungen an der Geschäftslogik können zentral in der Datenbank vorgenommen werden, ohne dass die Anwendung geändert werden muss.

- **Portabilität:** Da Stored Procedures in einer speziellen prozeduralen Sprache geschrieben sind, die von der Datenbank abhängig ist (z. B. PL/pgSQL für PostgreSQL), kann es schwierig sein, sie zwischen verschiedenen Datenbanksystemen zu migrieren. Man müsste möglicherweise den Code an die Syntax und Funktionen der anderen Datenbanksysteme anpassen.
- **Fehlende Versionskontrolle:** Im Gegensatz zu Anwendungscode, der in einem Versionskontrollsystem gespeichert werden kann, ist die Versionskontrolle von Stored Procedures oft schwieriger. Die Anpassung an Änderungen und das Nachverfolgen der Historie können erschwert werden, insbesondere wenn mehrere Entwickler an den Prozeduren arbeiten.
- **Komplexität:** Da die Geschäftslogik auf den Datenbankservern ausgeführt wird, kann die Verwaltung der Datenbank und die Diagnose von Leistungsproblemen komplexer werden. Es kann schwierig sein, Performance-Engpässe zu identifizieren und zu beheben, insbesondere wenn Stored Procedures mehrere Schichten der Geschäftslogik enthalten.
- **Testbarkeit:** Die Testbarkeit von Stored Procedures kann eingeschränkt sein, insbesondere im Vergleich zu Anwendungscode, der leichter isoliert und getestet werden kann. Das Erstellen von Unit-Tests und das Debuggen von Stored Procedures kann schwieriger und zeitaufwendiger sein.
- **Entwicklungsumgebung:** Entwickler müssen möglicherweise zusätzliche Tools und Techniken erlernen, um Stored Procedures effektiv zu entwickeln, zu testen und zu debuggen. Dies kann die Lernkurve erhöhen und die Produktivität beeinträchtigen.

User-Defined Functions (UDF)

- Eine UDF (User-Defined Function) in Postgres ist eine benutzerdefinierte Funktion, die von einem Entwickler oder Benutzer geschrieben wurde, um spezielle Operationen oder Berechnungen durchzuführen, die nicht von den integrierten Funktionen des Datenbanksystems abgedeckt werden.
- UDFs in PostgreSQL ermöglichen es, komplexere Operationen und Datenmanipulationen direkt in der Datenbank auszuführen, ohne auf externe Anwendungen angewiesen zu sein. Sie können verwendet werden, um die Leistung von Abfragen zu optimieren, benutzerdefinierte Aggregatfunktionen zu erstellen oder komplexe Geschäftslogiken direkt innerhalb der Datenbank zu implementieren.
- Standard-Sprache ist PL/pgSQL. Es werden aber auch andere sogenannte Sandbox-Sprachen wie Python oder C unterstützt.

PL/pgSQL

PL/pgSQL steht für "Procedural Language/PostgreSQL Structured Query Language" und ist eine prozedurale Programmiersprache für das PostgreSQL-Datenbankmanagementsystem. Es ist eine Erweiterung von SQL, die es ermöglicht, komplexe Geschäftslogiken und Datenmanipulationen direkt innerhalb der PostgreSQL-Datenbank zu implementieren.



```
CREATE FUNCTION function_name(parameter1 data_type, parameter2 data_type,
... )
RETURNS return_data_type
LANGUAGE plpgsql
AS $$
DECLARE
    -- Variablendeklarationen hier
BEGIN
    -- Funktionslogik hier
END;
$$;
```

In PostgreSQL besteht die Syntax zum Erstellen einer benutzerdefinierten Funktion (UDF) im Allgemeinen aus:

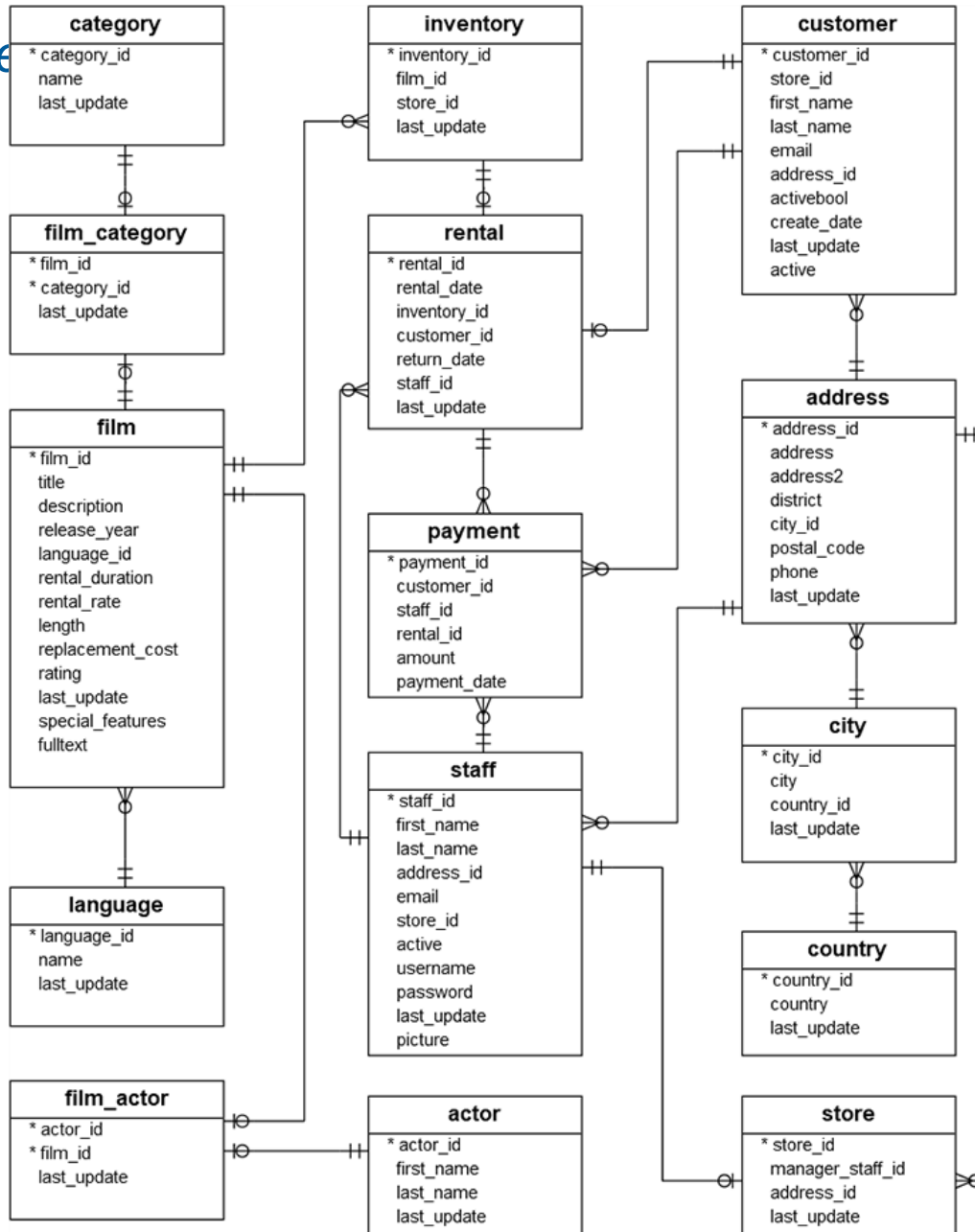
- **CREATE FUNCTION** Anweisung gefolgt von der Funktionsdefinition (inkl. Funktionsnamen, Eingangsparameter, Rückgabetyt)
- **AS \$\$... \$\$** -Syntax dient dazu, den Funktionskörper bzw. den PL/pgSQL-Code einzuschließen
- Das Schlüsselwort **DECLARE** wird verwendet, um Variablen und andere Objekte zu deklarieren, die im Funktionskörper verwendet werden









Anzeigen:

```
\df
```

Löschen:

```
DROP FUNCTION name
```



Entitätsmenge A	Entitätsmenge B	Mind.	Maximal
		1	viele
		1	1
		0	1
		0	viele

Frage: Wir wollen eine Funktion, die die Anzahl der Filme zählt die innerhalb einer definierten Range sind für die rental_rate (Tabelle film). 2 Parameter (min_rental_rate, max_rental_rate).


```

CREATE FUNCTION count_rr (min_r decimal(4,2), max_r decimal (4,2))
RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
movie_count INT;
BEGIN
SELECT COUNT(*)
INTO movie_count
FROM film
WHERE rental_rate BETWEEN min_r AND max_r;
RETURN movie_count;
END;
$$;


```



```
select count_rr(0,6)
```

	count_rr integer 
1	1000

```
select count_rr(3,10)
```

	count_rr integer 
1	336

Beispiel

```
CREATE FUNCTION count_rr(min_r decimal(4,2), max_r decimal(4,2))
RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
    movie_count INT;
BEGIN
    SELECT COUNT(*)
    INTO movie_count
    FROM film
    WHERE rental_rate BETWEEN min_r AND max_r;

    IF movie_count > 0 THEN
        RETURN movie_count;
    ELSE
        RETURN -1;
    END IF;
END;
$$;
```

Hinweis zu Fehlerbehandlung / Exceptions

Beispiel RAISE NOTICE

```
CREATE FUNCTION count_rr(min_r decimal(4,2), max_r decimal(4,2))
RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
    movie_count INT;
BEGIN
    SELECT COUNT(*)
    INTO movie_count
    FROM film
    WHERE rental_rate BETWEEN min_r AND max_r;

    IF movie_count = 0 THEN
        RAISE NOTICE 'Es gibt keine Filme in der angegebenen Preisspanne von % bis %.', min_r, max_r;
    END IF;

    RETURN movie_count;
END;
$$;
```

Hinweis zu Fehlerbehandlung / Exceptions

Beispiel RAISE EXCEPTION

```
CREATE FUNCTION count_rr(min_r decimal(4,2), max_r decimal(4,2))
RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
    movie_count INT;
BEGIN
    IF min_r < 0 OR max_r < 0 THEN
        RAISE EXCEPTION 'Negative Werte sind nicht erlaubt: %, %', min_r, max_r;
    END IF;

    SELECT COUNT(*)
    INTO movie_count
    FROM film
    WHERE rental_rate BETWEEN min_r AND max_r;

    IF movie_count = 0 THEN
        RAISE NOTICE 'Es gibt keine Filme in der angegebenen Preisspanne von % bis %.', min_r, max_r;
    END IF;

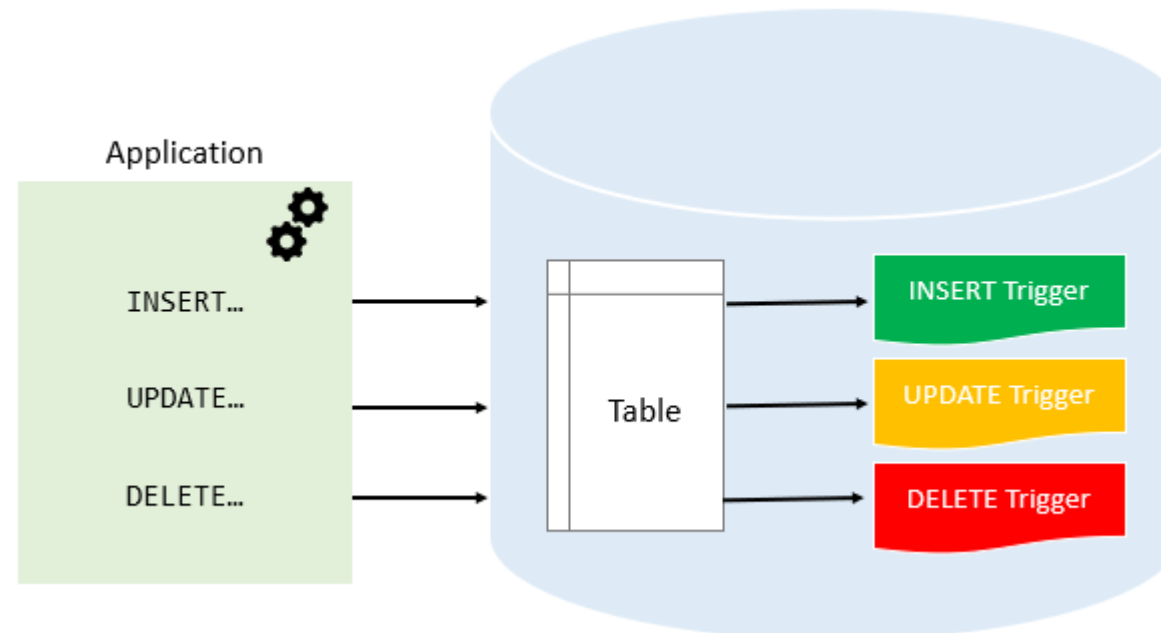
    RETURN movie_count;
END;
$$;
```


Exception	SQL Error Code	
NO_DATA_FOUND	+100	It is raised when a SELECT INTO statement returns no rows.
ZERO_DIVIDE	1476	It is raised when an attempt is made to divide a number by zero.
INVALID_NUMBER	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.

Database Trigger

Database Trigger

Ein Database Trigger ist eine spezielle Art von benutzerdefinierter Funktion oder Routine, die automatisch ausgeführt wird, wenn ein bestimmtes Ereignis in einer Datenbank auftritt. Diese Ereignisse sind in der Regel Datenmanipulationsoperationen wie INSERT, UPDATE, DELETE oder manchmal auch DDL-Operationen wie CREATE, ALTER und DROP.



Triggers werden häufig verwendet, um ...

- **Datenintegrität aufrechtzuerhalten:** Sie können Regeln und Überprüfungen durchführen, um sicherzustellen, dass die Datenkonsistenz erhalten bleibt, wenn Daten in einer Tabelle geändert werden.
- **Automatische Aktionen** durchführen: Triggers können automatisch zusätzliche Operationen ausführen, wenn Datenänderungen stattfinden, z. B. das Aktualisieren von Feldern, das Einfügen von Datensätzen in andere Tabellen oder das Löschen von Daten.
- **Protokollierung:** Triggers können verwendet werden, um Änderungen an Daten in Protokolltabellen aufzuzeichnen, die dazu beitragen, die Historie von Änderungen an Daten zu verfolgen und Aktivitäten in der Datenbank nachzuvollziehen.
- **Validierung:** Sie können Triggers verwenden, um bestimmte Validierungsregeln auf Daten anzuwenden, bevor sie in die Tabelle eingefügt oder aktualisiert werden, und Fehler zurückgeben, wenn die Regeln nicht eingehalten werden.
- **Benachrichtigungen:** In einigen Datenbanksystemen können Triggers verwendet werden, um Benachrichtigungen oder Warnungen zu senden, wenn bestimmte Ereignisse eintreten.



Es ist wichtig, Triggers sorgfältig und sparsam einzusetzen, da sie die Leistung beeinträchtigen und schwer zu debuggen sein können, wenn sie in großer Anzahl oder ohne Bedacht verwendet werden. Außerdem können sie das Verständnis und die Wartung der Datenbanklogik erschweren, da sie im Hintergrund arbeiten und nicht immer offensichtlich sind.

1. Schritt

Lege eine Trigger-Funktion an:

```
CREATE FUNCTION
trigger_function()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
AS $$
BEGIN
  -- trigger logic
END;
$$
```



2. Schritt

Binde die Trigger-Funktion an eine Tabelle mit dem `CREATE TRIGGER` Befehl:

```
CREATE TRIGGER trigger_name
  {BEFORE | AFTER} { event }
  ON table_name
  [FOR [EACH] { ROW | STATEMENT
  }]
      EXECUTE PROCEDURE
trigger_function
```

Beispiel: INSERT and UPDATE Triggers (1/4)

Anlage von zwei Relationen

```
DROP TABLE IF EXISTS employees;
```

```
CREATE TABLE employees (
    id INT GENERATED ALWAYS AS IDENTITY,
    first_name VARCHAR(40) NOT NULL,
    last_name VARCHAR(40) NOT NULL,
    PRIMARY KEY(id)
);
```

```
CREATE TABLE employee_audits (
    id INT GENERATED ALWAYS AS IDENTITY,
    employee_id INT NOT NULL,
    last_name VARCHAR(40) NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
);
```

Beispiel: INSERT and UPDATE Triggers (2/4)

Erstellen der Audit-Funktion

```
CREATE OR REPLACE FUNCTION log_last_name_changes()  
    RETURNS TRIGGER  
    LANGUAGE PLPGSQL  
    AS  
    $$  
BEGIN  
    IF NEW.last_name <> OLD.last_name THEN  
        INSERT INTO employee_audits(employee_id,last_name,changed_on)  
        VALUES (OLD.id,OLD.last_name,now());  
    END IF;  
  
    RETURN NEW;  
END;  
$;  
;
```


Beispiel: *INSERT and UPDATE Triggers (3/4)*

Erstellen des Triggers gebunden an die Tabelle employees

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Beispiel: *INSERT and UPDATE Triggers (4/4)*

Datensätze einfügen

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');
```

```
INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');
```

```
SELECT * FROM employees;
```

```
*****
```

```
UPDATE employees
SET last_name = 'Brown'
WHERE ID = 2;
```

```
SELECT * FROM employee_audits;
```

Anzeigen:

`\dS table_name`

Großes S!



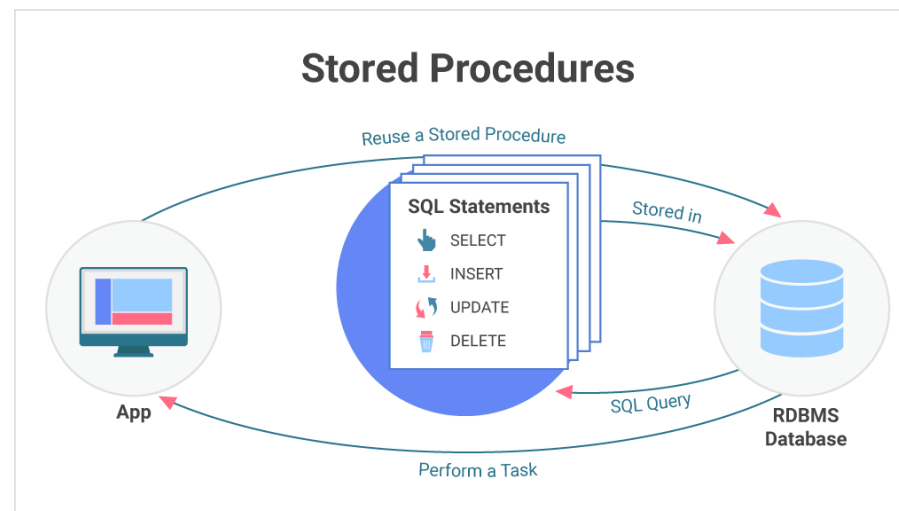
Löschen:

```
DROP TRIGGER [IF EXISTS] trigger_name ON table_name
```

Stored Procedures

Definition Stored Procedures (SP)

Eine Stored Procedure ist ein gespeichertes Programm (Skript) in einer Datenbank, das aus einer oder mehreren SQL-Anweisungen und optional zusätzlicher Programmierlogik besteht. Stored Procedures werden in der Datenbank selbst gespeichert und können von Anwendungen oder Benutzern aufgerufen werden, um komplexe Datenbankoperationen auszuführen oder Geschäftslogik zu implementieren. Hauptunterschied: SP unterstützen Transaktionen und können keinen Datentyp zurückgeben.



```
CREATE PROCEDURE <procedure_name> (param1, param2,...)
LANGUAGE plpgsql
AS
$$
DECLARE
<variable declaration>
BEGIN
<procedure_definition>
END;
$$
```

„One Unit of Work“

id [PK] integ	first_name text	last_name text	amount numeric (9,2)
1	Tim	Brown	2500.00
2	Sandra	Miller	1600.00



Überweisung 100

```
CREATE PROCEDURE sp_transfer (tr_amount INT, sender INT, recipient INT)
LANGUAGE plpgsql
AS
$$
BEGIN
    -- subtract from sender's balance
    UPDATE acc_balance
    SET amount = amount - tr_amount
    WHERE id = sender;
    -- add to recipient's balance
    UPDATE acc_balance
    SET amount = amount + tr_amount
    WHERE id = recipient;
    COMMIT;
END;
$$
```

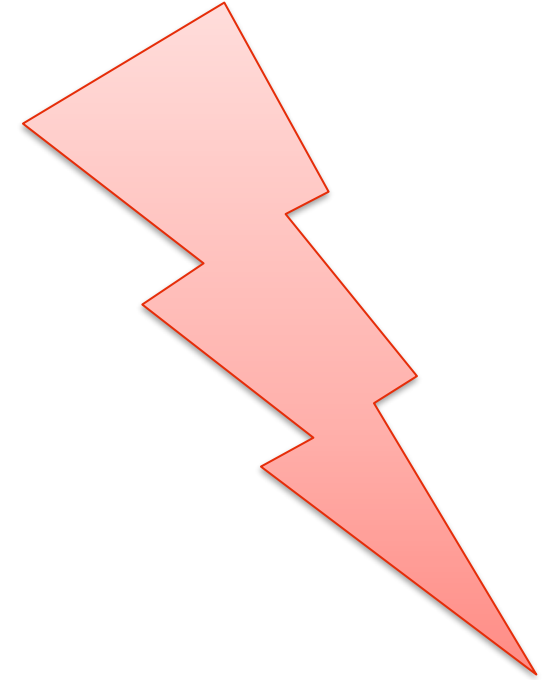


```
CALL <store_procedure_name> (param1, param2,...);
```

```
CALL sp_transfer (100, 1,2);
```

CTEs (Common Table Expressions)

```
select facid, sum(slots) as summe
from bookings
Group BY facid
having summe > 10
```



```
select facid, sum(slots) as summe  
from bookings  
Group BY facid  
having sum(slots) > 10
```

```
WITH sum_slots AS (  
    SELECT facid, SUM(slots) AS summe  
    FROM bookings  
    GROUP BY facid  
)  
SELECT facid, summe  
FROM sum_slots  
WHERE summe > 10;
```

CTEs (Common Table Expressions)

CTE steht für Common Table Expression. Es handelt sich um eine temporäre Ergebnismenge, die innerhalb einer einzelnen SQL-Anweisung definiert und verwendet wird. CTEs bieten eine Möglichkeit, komplexe Abfragen in übersichtlichere, leichter verständliche Abschnitte zu unterteilen. Sie sind besonders nützlich, um wiederkehrende Unterabfragen zu vermeiden und die Lesbarkeit des SQL-Codes zu verbessern.

```
-- Angenommen, die Tabelle "employees" hat die Spalten "id", "name", "department"

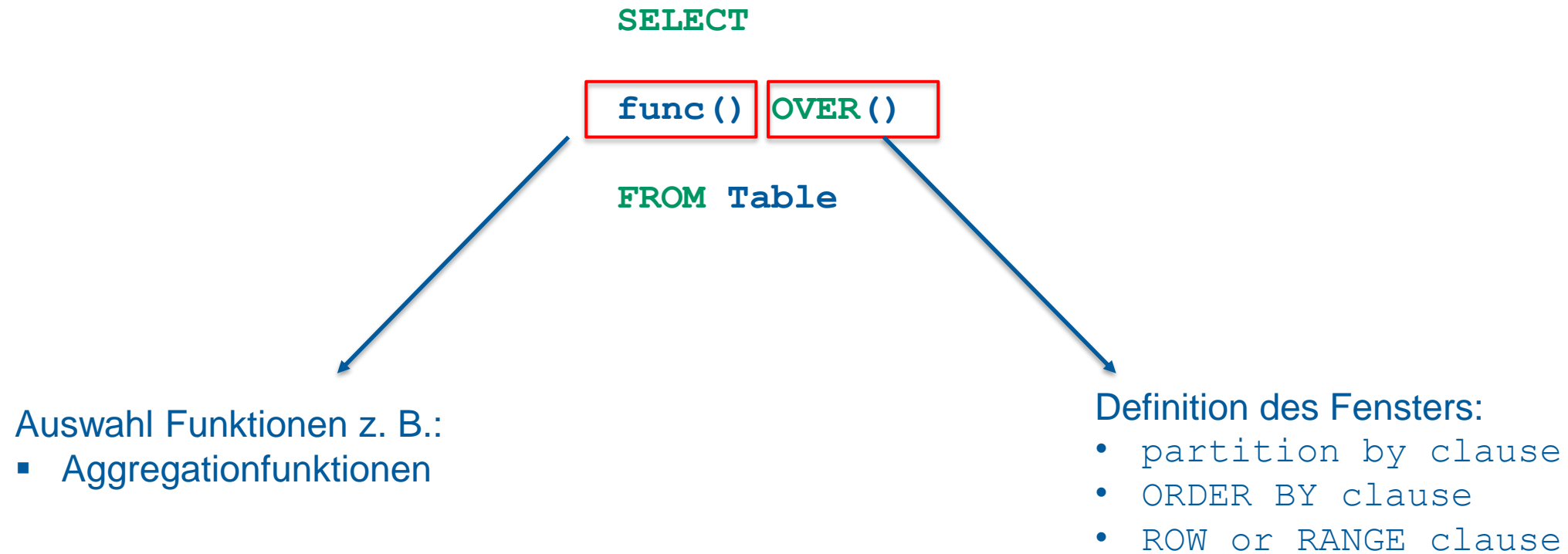
WITH department_counts AS (
    SELECT
        department,
        COUNT(id) AS employee_count
    FROM
        employees
    GROUP BY
        department
)
SELECT
    department,
    employee_count
FROM
    department_counts
ORDER BY
    employee_count DESC;
```

Window Functions

Definition

Eine Window Function (Fensterfunktion) ist eine spezielle Art von Funktion in SQL, die es ermöglicht, Berechnungen über eine Gruppe von Zeilen durchzuführen, die in Beziehung zur aktuellen Zeile stehen. Im Gegensatz zu herkömmlichen Aggregatfunktionen, die eine einzelne Wertezusammenfassung für eine Gruppe von Zeilen erstellen, behält eine Fensterfunktion die individuellen Zeilen bei und fügt zusätzliche Informationen hinzu, die auf der berechneten Gruppe basieren. Fensterfunktionen sind besonders nützlich, wenn Sie laufende Summen, gleitende Durchschnitte oder Rangfolgen innerhalb einer Tabelle berechnen möchten.







Day	Miles Driving
Jan. 1	60
Jan. 2	80
Jan. 3	10
Jan. 4	150

```
SELECT Day, Mile Driving,  
SUM(Miles Driving) OVER(ORDER BY Day) AS  
Running_Total  
FROM table;
```

Running total mileage visual		
Day	Miles Driving	Running Total
Jan. 1	60	
Jan. 2	80	
Jan. 3	10	
Jan. 4	150	

Window Function am Beispiel

Berechnung des Durchschnitts der letzten drei Tage

Day	Daily_Revenue
1	39
2	528
3	39
4	86
5	86
6	351

```
SELECT Day, Daily_Revenue,
AVG(Daily_Revenue) OVER (ORDER BY Day ROWS
2 PRECEDING) AS Three_Day_Average
FROM revenue;
```

Running Average Example		
Day	Daily Revenue	3 Day Average
1	39	
2	528	
3	39	
4	86	
5	86	
6	351	

Window Function am Beispiel

Partitionierung des Fensters nach Wochenende

Day	Weekend	Daily_Revenue
1	T	528
7	T	867
2	F	47
3	F	39
4	F	86
5	F	86
6	F	351

```
SELECT Day, Weekend, Daily_Revenue,
SUM(Daily_Revenue) OVER(PARTITION BY
Weekend ORDER BY Weekend DESC) AS Total
FROM revenue2
```

Partitioned Total Example			
Day	Weekend	Daily Revenue	Total
1	T	528	
2	F	47	
3	F	39	
4	F	86	
5	F	86	
6	F	351	
7	T	867	

Partitionierung von Tabellen

Definition

Partitionierung in Postgres ist eine Technik zur Verbesserung der Abfrageleistung und Verwaltung großer Tabellen, indem die Daten in kleinere, logisch getrennte und leichter handhabbare Teile (Partitionen) aufgeteilt werden. Diese Partitionen basieren auf bestimmten Kriterien wie Wertebereichen oder Diskreten Werten von ausgewählten Spalten. Die Abfragen greifen dann nur auf die relevanten Partitionen zu, was die Leistung erhöht.

Erst ab > 1 Millionen Zeilen empfohlen!

Orders table
Not Clustered; Not partitioned

Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-04	JP	Shipped
2022-08-05	UK	Canceled
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-05	US	Processing
2022-08-04	JP	Processing
2022-08-04	KE	Shipped
2022-08-06	UK	Canceled
2022-08-02	UK	Processing
2022-08-05	JP	Canceled
2022-08-06	UK	Processing
2022-08-05	US	Shipped
2022-08-06	JP	Processing
2022-08-02	KE	Shipped
2022-08-04	US	Shipped

Orders table
Clustered by Country; Not partitioned

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	JP	Processing
2022-08-05	JP	Canceled
2022-08-06	JP	Processing
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-04	KE	Shipped
2022-08-02	KE	Shipped
2022-08-05	UK	Processing
2022-08-06	UK	Canceled
2022-08-02	UK	Canceled
2022-08-06	UK	Processing
2022-08-02	US	Shipped
2022-08-05	US	Processing
2022-08-05	US	Shipped
2022-08-04	US	Shipped

Orders table
Clustered by Country; Partitioned by Order_Date (Daily)

	Order_Date	Country	Status
Partition:	2022-08-02	KE	Shipped
	2022-08-02	KE	Canceled
Clusters:	2022-08-02	UK	Processing
Country	2022-08-02	US	Shipped

	Order_Date	Country	Status
Partition:	2022-08-04	JP	Shipped
	2022-08-04	JP	Processing
Cluster:	2022-08-04	KE	Shipped
Country	2022-08-04	US	Shipped

	Order_Date	Country	Status
Partition:	2022-08-05	JP	Canceled
	2022-08-05	UK	Canceled
Cluster:	2022-08-05	US	Shipped
Country	2022-08-05	US	Processing

	Order_Date	Country	Status
Partition:	2022-08-06	JP	Processing
	2022-08-06	KE	Shipped
Cluster:	2022-08-06	UK	Canceled
Country	2022-08-06	UK	Processing

- **Range Partitioning:** Partitionierung einer Tabelle nach einem Wertebereich. Dies wird häufig bei Datumsfeldern verwendet, z. B. bei einer Tabelle mit Verkaufsdaten, die in monatliche Partitionen unterteilt ist, entsprechend dem Verkaufsdatum
- **List Partitioning:** Partitionierung einer Tabelle nach einer Liste bekannter Werte. Dies wird typischerweise verwendet, wenn der Partitions-Schlüssel einen kategorialen Wert hat, z. B. bei einer globalen Verkaufstabelle, die in regionale Partitionen unterteilt ist. Der Partitions-Schlüssel kann in diesem Fall der Länder- oder Städtecode sein, und jede Partition definiert die Liste der Codes, die ihr zugeordnet sind.
- **Hash Partitioning:** Partitionierung einer Tabelle mit einer Hash-Funktion auf dem Partitions-Schlüssel. Dies ist besonders nützlich, wenn es keine offensichtliche Möglichkeit gibt, Daten in logisch ähnliche Gruppen aufzuteilen, und wird häufig bei kategorialen Partitionsschlüsseln verwendet, auf die individuell zugegriffen wird. Z. B., wenn eine Verkaufstabelle häufig nach Produkt abgefragt wird, könnte die Tabelle von einer Hash-Partitionierung auf der Produkt-SKU profitieren.

1. Schritt

Tabelle anlegen und
Partitionierungsmethode spezifizieren

```
CREATE TABLE table_name (  
    column1 data_type1,  
    column2 data_type2,  
    ...  
) PARTITION BY partition_type  
    (partition_column);
```

2. Schritt

Wertebereiche definieren

```
CREATE TABLE partition_table_name  
    PARTITION OF table_name FOR  
    VALUES partition_criteria;
```



```
CREATE TABLE orders (  
    id SERIAL,  
    order_date DATE NOT NULL,  
    customer_id INTEGER,  
    amount NUMERIC(10, 2),  
    PRIMARY KEY (order_date, id)  
) PARTITION BY RANGE (order_date);
```

```
CREATE TABLE orders_2020 PARTITION OF  
orders FOR VALUES FROM ('2020-01-01') TO  
('2021-01-01');
```

```
CREATE TABLE orders_2021 PARTITION OF  
orders FOR VALUES FROM ('2021-01-01') TO  
('2022-01-01');
```

```
CREATE TABLE orders_2022 PARTITION OF  
orders FOR VALUES FROM ('2022-01-01') TO  
('2023-01-01');
```



```
CREATE TABLE orders (  
    id SERIAL PRIMARY KEY,  
    order_date DATE NOT NULL,  
    customer_id INTEGER,  
    amount NUMERIC(10, 2)  
) PARTITION BY LIST (id);
```

```
CREATE TABLE orders_group1 PARTITION OF orders FOR VALUES IN (1,  
2, 3, 4, 5);
```

```
CREATE TABLE orders_group2 PARTITION OF orders FOR VALUES IN (6,  
7, 8, 9, 10);
```

```
CREATE TABLE orders_group3 PARTITION OF orders FOR VALUES IN  
(11, 12, 13, 14, 15);
```

Beispiel SQL

Hash Partitioning



```
CREATE TABLE orders (  
    id SERIAL PRIMARY KEY,  
    order_date DATE NOT NULL,  
    customer_id INTEGER,  
    amount NUMERIC(10, 2)  
) PARTITION BY HASH (id);
```

```
CREATE TABLE orders_part0 PARTITION OF orders FOR  
VALUES WITH (MODULUS 4, REMAINDER 0);
```

```
CREATE TABLE orders_part1 PARTITION OF orders FOR  
VALUES WITH (MODULUS 4, REMAINDER 1);
```

```
CREATE TABLE orders_part2 PARTITION OF orders FOR  
VALUES WITH (MODULUS 4, REMAINDER 2);
```

```
CREATE TABLE orders_part3 PARTITION OF orders FOR  
VALUES WITH (MODULUS 4, REMAINDER 3);
```

DROP TABLE orders löscht auch die Partitionen mit!