MySQL - Zusammenfassung

SELECT & FROM

Das `SELECT` Schluesselwort erlaubt es einem aus Daten aus einer oder mehreren Datenbanktabellen auszuwählen

```
SELECT select_list
FROM table_name;
```

Nach dem `SELECT` kommt ein oder mehrere Datensätze die man auswählen will Mit dem `FROM` kann man auswählen aus welcher Tabelle man die Daten lesen will

Bsp1 `SELECT FROM` um Daten von einer einzelnen Spalte zu erhalten:

```
SELECT lastName
FROM employees;
```

OUTPUT:

```
+-----------+
| lastName  |
+-----------+
| Murphy    |
| Patterson |
| Firrelli  |
| Patterson |
| Bondur    |
| Bow       |
| Jennings  |
...
```

Bsp2 `SELECT FROM` um Daten von mehreren Spalten zu erhalten:

```
SELECT
    lastName,
    firstName,
    jobTitle
FROM
    employees;
```

OUTPUT:

```
+-----------+-----------+----------------------+
| lastname  | firstname | jobtitle             |
+-----------+-----------+----------------------+
| Murphy    | Diane     | President            |
| Patterson | Mary      | VP Sales             |
| Firrelli  | Jeff      | VP Marketing         |
| Patterson | William   | Sales Manager (APAC) |
| Bondur    | Gerard    | Sale Manager (EMEA)  |
...
```

ORDER BY

Der `ORDER BY` Befehl ist dazu da um die Spalten des Ergebnisses zu sortieren.

```
SELECT
    select_list
FROM
    table_name
ORDER BY
    column1 [ASC|DESC],
    column2 [ASC|DESC],
    ...;
```

`ASC` steht für aufsteigend sortieren, während `DESC` für absteigend sortieren

Bsp:

```
SELECT
    contactLastname,
    contactFirstname
FROM
    customers
ORDER BY
    contactLastname;
```

OUTPUT:

```
+-----------------+------------------+
| contactLastname | contactFirstname |
+-----------------+------------------+
| Accorti         | Paolo            |
| Altagar,G M     | Raanan           |
| Andersen        | Mel              |
| Anton           | Carmen           |
| Ashworth        | Rachel           |
```

```
| Barajas          | Miguel           |
...
```

Hier wurde in aufsteigender Reihenfolge nach dem Nachnamen sortiert.

So würde absteigend sortiert werden:

```
SELECT
    contactLastname,
    contactFirstname
FROM
    customers
ORDER BY
    contactLastname DESC;
```

OUTPUT:

```
+-----------------+------------------+
| contactLastname | contactFirstname |
+-----------------+------------------+
| Young           | Jeff             |
| Young           | Julie            |
| Young           | Mary             |
| Young           | Dorothy          |
| Yoshido         | Juri             |
| Walker          | Brydey           |
| Victorino       | Wendy            |
| Urs             | Braun            |
| Tseng           | Jerry            |
....
```

Man kann auch nach mehreren Spalten sortieren:

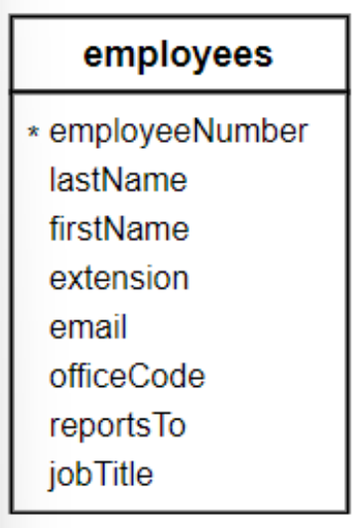
```
SELECT
    contactLastname,
    contactFirstname
FROM
    customers
ORDER BY
    contactLastname DESC ,
    contactFirstname ASC;
```

OUTPUT:

```
+-----------------+------------------+
| contactLastname | contactFirstname |
+-----------------+------------------+
| Young           | Dorothy          |
| Young           | Jeff             |
| Young           | Julie            |
| Young           | Mary             |
| Yoshido         | Juri             |
| Walker          | Brydey           |
| Victorino       | Wendy            |
| Urs             | Braun            |
| Tseng           | Jerry            |
| Tonini          | Daniel           |
...
```

WHERE

Mit dem `WHERE` Befehl kann man man nach gewissen eigenschaften eines Datensatzes Filtern, wie z.B: einer Jobbezeichnung:

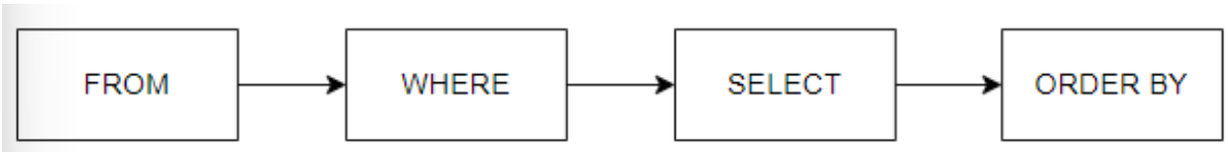



```sql
SELECT
    lastname,
    firstname,
    jobtitle
FROM
    employees
WHERE
    jobtitle = 'Sales Rep';
```

OUTPUT:

```
+-----------+-----------+-----------+
| lastname  | firstname | jobtitle  |
+-----------+-----------+-----------+
| Jennings  | Leslie    | Sales Rep |
| Thompson  | Leslie    | Sales Rep |
| Firrelli  | Julie     | Sales Rep |
| Patterson | Steve     | Sales Rep |
| Tseng     | Foon Yue  | Sales Rep |
| Vanauf    | George    | Sales Rep |
| Bondur    | Loui      | Sales Rep |
| Hernandez | Gerard    | Sales Rep |
| Castillo  | Pamela    | Sales Rep |
| Bott      | Larry     | Sales Rep |
| Jones     | Barry     | Sales Rep |
| Fixter    | Andy      | Sales Rep |
| Marsh     | Peter     | Sales Rep |
| King      | Tom       | Sales Rep |
| Nishi     | Mami      | Sales Rep |
| Kato      | Yoshimi   | Sales Rep |
| Gerard    | Martin    | Sales Rep |
+-----------+-----------+-----------+
```

Reihenfolge:



Man kann das `WHERE` Argument auch mit `AND` kombinieren:

```sql
SELECT
    lastname,
    firstname,
    jobtitle,
    officeCode
FROM
    employees
WHERE
    jobtitle = 'Sales Rep' AND
    officeCode = 1;
```

OUTPUT:

```
+----------+-----------+-----------+------------+
| lastname | firstname | jobtitle  | officeCode |
+----------+-----------+-----------+------------+
```

```
| Jennings | Leslie    | Sales Rep | 1          |
| Thompson | Leslie    | Sales Rep | 1          |
+----------+-----------+-----------+------------+
```

Ein weiterer wichtiger Befehl in kombination mit `WHERE` ist `LIKE`. Hier kann man Strings vergleichen. Wenn man zum Beispiel alle Mitarbeiter einer Firma finden will, dessen Nachname mit *son* aufhört, kann man das so machen:

```
SELECT
    firstName,
    lastName
FROM
    employees
WHERE
    lastName LIKE '%son'
ORDER BY firstName;
```

OUTPUT:

```
+-----------+-----------+
| firstName | lastName  |
+-----------+-----------+
| Leslie    | Thompson  |
| Mary      | Patterson |
| Steve     | Patterson |
| William   | Patterson |
+-----------+-----------+
```

Ebenfalls wichtig ist der `IN` Befehl, welcher einfach nur überprüft ob ein Wert mit etwas übereinstimmt:

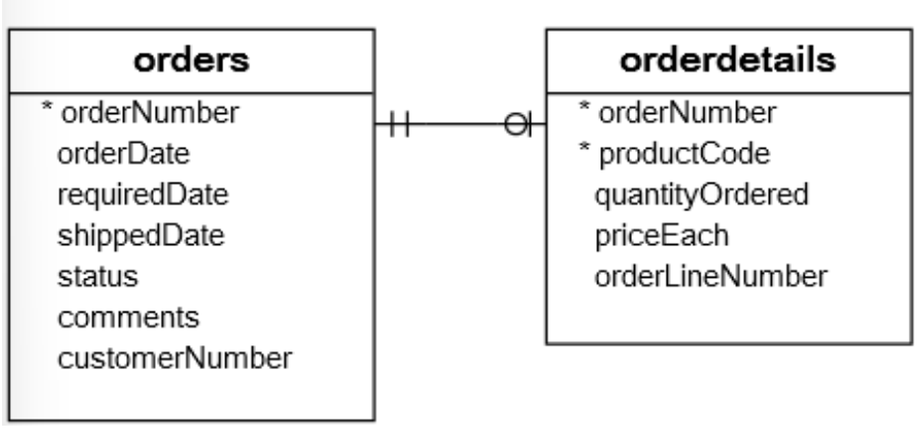
```
SELECT
    firstName,
    lastName,
    officeCode
FROM
    employees
WHERE
    officeCode IN (1 , 2, 3)
ORDER BY
    officeCode;
```

OUTPUT:

```
+-----------+-----------+------------+
| firstName | lastName  | officeCode |
+-----------+-----------+------------+
| Diane     | Murphy    | 1          |
| Mary      | Patterson | 1          |
| Jeff      | Firrelli  | 1          |
| Anthony   | Bow       | 1          |
| Leslie    | Jennings  | 1          |
| Leslie    | Thompson  | 1          |
| Julie     | Firrelli  | 2          |
| Steve     | Patterson | 2          |
| Foon Yue  | Tseng     | 3          |
| George    | Vanauf    | 3          |
+-----------+-----------+------------+
```

JOINS

In vielen Tabellen hängen Daten, welche in verschiedenen Tabellen sind zusammen. Um alle Zusammenhängende Daten zu bekommen, kann man verschiedene Arten von `JOINS` verwenden.



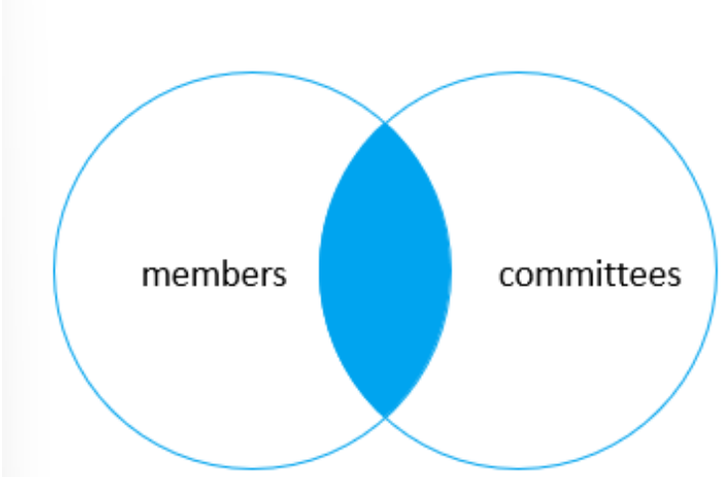
Verschiedene Arten vond `JOINS`:

```sql
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
INNER JOIN committees c ON c.name = m.name;
```

OUTPUT:

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
```

```
+-----------+--------+-------------+-----------+
|         1 | John   |           1 | John      |
|         3 | Mary   |           2 | Mary      |
|         5 | Amelia |           3 | Amelia    |
+-----------+--------+-------------+-----------+
```



Wenn beide Attribute gleich sind kann man auch `USING` verwenden um es etwas kürzer zu halten:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
INNER JOIN committees c USING(name);
```

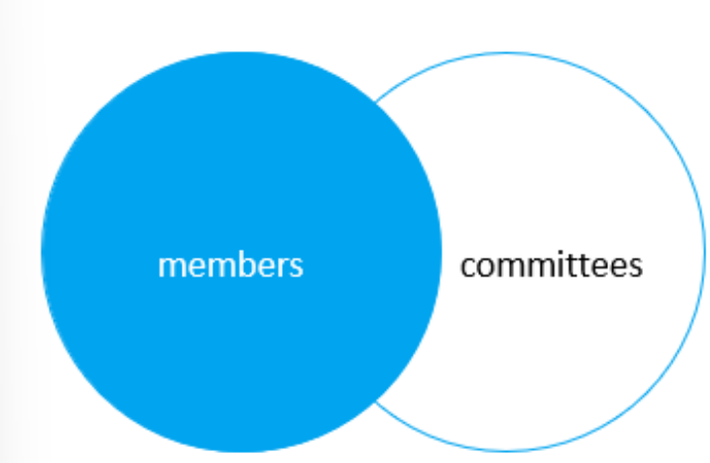
LEFT JOIN:

Bei einem `LEFT JOIN` werden die Tabellen in eine "linke" und "rechte" Tabelle aufgeteilt. Jetzt werden die Daten von der "linken" Tabelle mit der "rechten" verglichen. Auch wenn die Werte nicht übereinstimmen, werden trotzdem neue Zeilen erstellt in welchen die Werte der "linken" Tabelle gespeichert werden und für die Werte der "rechten" wird *NULL* eingesetzt. Im Endeffekt werden bei einem `LEFT JOIN` immer alle Werte der "linken" Tabelle ausgewählt, egal ob sie mit der "rechten" Tabelle übereinstimmen.

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
LEFT JOIN committees c USING(name);
```

OUTPUT:

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         2 | Jane   |         NULL | NULL      |
|         3 | Mary   |            2 | Mary      |
|         4 | David  |         NULL | NULL      |
|         5 | Amelia |            3 | Amelia    |
+-----------+--------+--------------+-----------+
```



Kann auch wieder mit `USING` verwendet werden:

```sql
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
LEFT JOIN committees c USING(name);
```

Einen `LEFT JOIN` kann man auch nutzen um zum Beispiel jede Zeile herauszufiltern, in welcher der zu vergleichende Wert *NULL* oder nicht *NULL* ist.

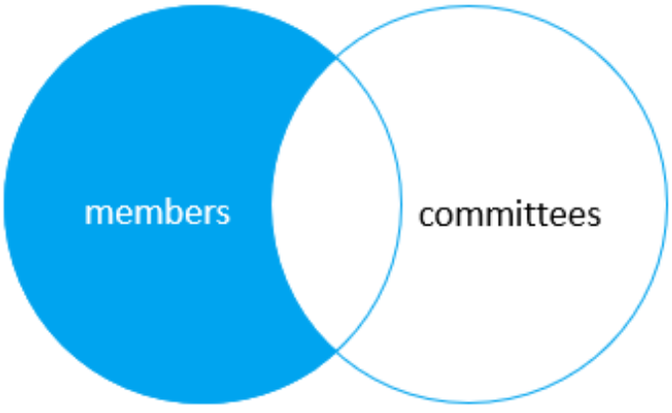
So sieht man alle Zeilen in welchen der Wert der "rechten" Tabelle *NULL* ist, bzw. in welcher die Werte nur in der "linken" Tabelle gespeichert sind:

```sql
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
```

```
    c.name AS committee
FROM
    members m
LEFT JOIN committees c USING(name)
WHERE c.committee_id IS NULL;
```

OUTPUT:

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         2 | Jane   |         NULL | NULL      |
|         4 | David  |         NULL | NULL      |
+-----------+--------+--------------+-----------+
```



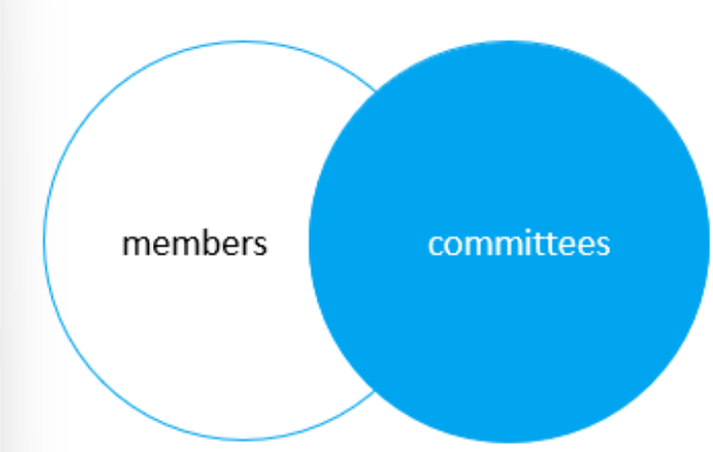
RIGHT JOIN

Wie man beim Namen `RIGHT JOIN` schon vermuten kann funktioniert er genau gleich wie der `LEFT JOIN` nur invertiert. Es werden also alle Zeilen von der "rechten" Tabelle mit der "linken" verglichen. Wenn etwas nicht übereinstimmt, werden die Werte für die "linke" Tabelle mit *NULL* ersetzt.

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
RIGHT JOIN committees c on c.name = m.name;
```

OUTPUT:

```
+-----------+--------+-------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+-------------+-----------+
|         1 | John   |            1 | John      |
|         3 | Mary   |            2 | Mary      |
|         5 | Amelia |            3 | Amelia    |
|      NULL | NULL   |            4 | Joe       |
+-----------+--------+-------------+-----------+
```



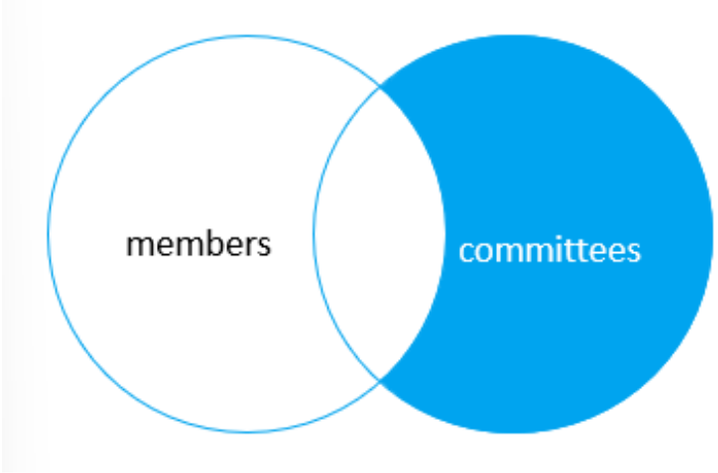
Natürlich kann man auch bei dem `RIGHT JOIN` wieder `USING` verwenden:

So kann man zum Beispiel alle Zeilen herausfiltern, in welchen der Wert der "linken" Tabelle *NULL* ist, also in welcher nur die Werte der "rechten" Tabelle gespeichert sind:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
RIGHT JOIN committees c USING(name)
WHERE m.member_id IS NULL;
```

OUTPUT:

```
+-----------+--------+-------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+-------------+-----------+
|      NULL | NULL   |            4 | Joe       |
+-----------+--------+-------------+-----------+
```



Cross Join

Der `CROSS JOIN` ist der einfachste `JOIN` von allen. Er verbindet einfach jede Zeile der "linken" Tabelle mit jeder Zeile der "rechten" Tabelle.

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
CROSS JOIN committees c;
```

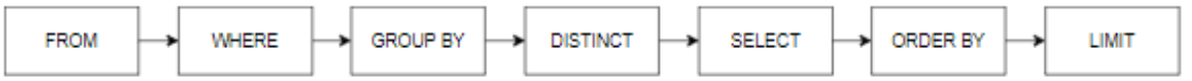
OUTPUT:

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            4 | Joe       |
|         1 | John   |            3 | Amelia    |
|         1 | John   |            2 | Mary      |
|         1 | John   |            1 | John      |
|         2 | Jane   |            4 | Joe       |
|         2 | Jane   |            3 | Amelia    |
|         2 | Jane   |            2 | Mary      |
|         2 | Jane   |            1 | John      |
|         3 | Mary   |            4 | Joe       |
|         3 | Mary   |            3 | Amelia    |
|         3 | Mary   |            2 | Mary      |
|         3 | Mary   |            1 | John      |
|         4 | David  |            4 | Joe       |
|         4 | David  |            3 | Amelia    |
|         4 | David  |            2 | Mary      |
|         4 | David  |            1 | John      |
|         5 | Amelia |            4 | Joe       |
```

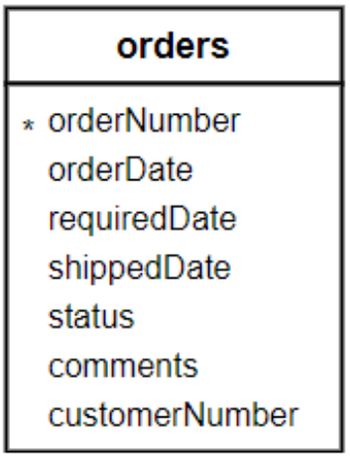
```
|         5 | Amelia |            3 | Amelia    |
|         5 | Amelia |            2 | Mary      |
|         5 | Amelia |            1 | John      |
+-----------+--------+--------------+-----------+
```

GROUP BY

Der `GROUP BY` Befehl wird verwendet um Daten zu gruppieren. `GROUP BY` wird nach dem `WHERE` Befehl ausgeführt.



Für das Beispiel nehmen wir die Tabelle `orders`:



Wenn man jetzt nach dem `status` gruppieren wil:

```sql
SELECT
  status
FROM
  orders
GROUP BY
  status;
```

OUTPUT:

```
+------------+
| status     |
+------------+
| Shipped    |
| Resolved   |
| Cancelled  |
| On Hold    |
| Disputed   |
```

```
| In Process |
+------------+
```

GROUP BY mit Aggregatfunktionen:

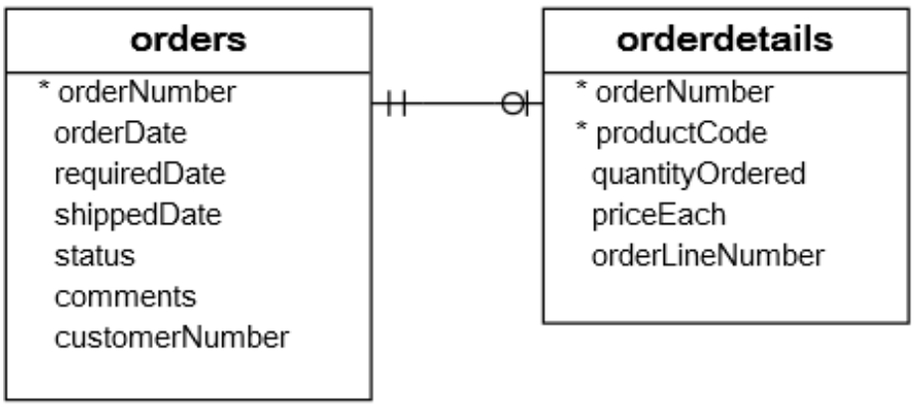
Meist wird `GROUP BY` mit Aggregatfunktionen wie `COUNT`, `SUM`, `AVG`, `MIN` oder `MAX` verwendet:

```
SELECT
  status,
  COUNT(*)
FROM
  orders
GROUP BY
  status;
```

OUTPUT:

```
+------------+----------+
| status     | COUNT(*) |
+------------+----------+
| Shipped    |      303 |
| Resolved   |        4 |
| Cancelled  |        6 |
| On Hold    |        4 |
| Disputed   |        3 |
| In Process |        6 |
+------------+----------+
```

Natürlich kann man `GROUP BY` auch mit mehreren Spalten verwenden:



Um alle Bestellungen nach dem status zu bekommen, wird die `orders` Tabelle mit der `orderdetails` Tabelle gejoint und die `SUM` Funktion verwendet um die Anzahl der Bestellungen zu zählen:

```
SELECT
  status,
  SUM(quantityOrdered * priceEach) AS amount
FROM
  orders
  INNER JOIN orderdetails USING (orderNumber)
GROUP BY
  status;
```

OUTPUT:

```
+------------+------------+
| status     | amount     |
+------------+------------+
| Shipped    | 8865094.64 |
| Resolved   |  134235.88 |
| Cancelled  |  238854.18 |
| On Hold    |  169575.61 |
| Disputed   |   61158.78 |
| In Process |  135271.52 |
+------------+------------+
```

GROUP BY nach einem Ausdruck:

Man kann mit `GROUP BY` auch nach Ergebnissen eines Ausdrucks und nicht einfach nach Werten einer Spalte gruppieren:

```
SELECT
  YEAR(orderDate) AS year,
  SUM(quantityOrdered * priceEach) AS total
FROM
  orders
  INNER JOIN orderdetails USING (orderNumber)
WHERE
  status = 'Shipped'
GROUP BY
  YEAR(orderDate);
```

OUTPUT:

```
+------+------------+
| year | total      |
+------+------------+
| 2003 | 3223095.80 |
| 2004 | 4300602.99 |
```

```
| 2005 | 1341395.85 |
+------+------------+
```

GROUP BY mit HAVING:

Um die Gruppierten Ergebnisse zu filtern wir `HAVING` verwendet:

```
SELECT
  YEAR(orderDate) AS year,
  SUM(quantityOrdered * priceEach) AS total
FROM
  orders
  INNER JOIN orderdetails USING (orderNumber)
WHERE
  status = 'Shipped'
GROUP BY
  year
HAVING
  year > 2003;
```

OUTPUT:

year	total
2004	4300602.99
2005	1341395.85

GROUP BY mit mehreren Spalten:

```
SELECT
  YEAR(orderDate) AS year,
  status,
  SUM(quantityOrdered * priceEach) AS total
FROM
  orders
  INNER JOIN orderdetails USING (orderNumber)
GROUP BY
  year,
  status
ORDER BY
  year;
```

OUTPUT:

```
+------+------------+------------+
| year | status     | total      |
+------+------------+------------+
```

```
| 2003 | Cancelled  |   67130.69 |
| 2003 | Resolved   |   27121.90 |
| 2003 | Shipped    | 3223095.80 |
| 2004 | Cancelled  |  171723.49 |
| 2004 | On Hold    |   23014.17 |
| 2004 | Resolved   |   20564.86 |
| 2004 | Shipped    | 4300602.99 |
| 2005 | Disputed   |   61158.78 |
| 2005 | In Process |  135271.52 |
| 2005 | On Hold    |  146561.44 |
| 2005 | Resolved   |   86549.12 |
| 2005 | Shipped    | 1341395.85 |
+------+------------+------------+
```

STORED PROCEDURES

`STORED PROCEDURES` sind Skripte die in der Datenbank gespeichert und ausgeführt werden können. Nützlich für wiederkehrende Aufgaben, welche komplexe Abfragen enthalten.

So wird eine `PROCEDURE` erstellt:

```
DELIMITER $$

CREATE PROCEDURE GetCustomers()
BEGIN
    SELECT
        customerName,
        city,
        state,
        postalCode,
        country
    FROM
        customers
    ORDER BY customerName;
END$$
DELIMITER ;
```

So wird sie aufgerufen:

```
CALL GetCustomers();
```

Vorteile:

- Da man bei einer `STORED PROCEDURE` nur einen Befehlaufrufen muss, wird der Netzwerkverkeht verringert
- Wiederverwendbarkeit

- Man kann die Datenbank sicherer machen, da man gewisse Aplikationen zugriff auf gewisse `STORED PROCEDURES` geben kann und anderern nicht.

Nachteile:

- Wenn viele `PROCEDURES` verwendet werden wird die Speichernutzung jeder Verbindung signifikant ansteigen. Außerdem wird die CPU - Auslastung ansteigen, wenn eine hohe Anzahl an logischen Operationen verwendet wird
- Da in `MySQL` einige Debugging Werkzeuge fehlen, ist es sehr anspruchsvoll `STORED PROCEDURES` zu debuggen.
- Wenn mit `STORED PROCEDURES` gearbeitet wird, gibt es oft Probleme in der Instandhaltung / Wartung, da nicht jeder Entwicker mit dem Konzept vertraut ist

DELIMITER:

Der `DELIMITER` ist das Symbol, das verwendet wird um Argumente zu trennen. Standardmäßig: `;`

Man kann den `DELIMITER` auch ändern:

```
DELIMITER delimiter_character
```

Da bei `STORED PROCEDURES` meist mehrere `;` vorkommen, muss man den Delimiter ändern, um die ganze Stored Procedure als ein Objekt zu kompilieren.

DROP:

Mit dem `DROP` Befehl kann man eine bereits erstellte `Procedure` löschen:

```
DROP PROCEDURE [IF EXISTS] sp_name;
```

Mann kann das `DROP` Argument auch noch mit anderen Argumenten wie z.B einem `IF` kombinieren.

LISTING STORED PROCEDURES:

Mit dem `SHOW PROCEDURE STATUS` kann man alle Karakteristiken eines `PROCEDURES` anzeigen lassen.

Syntax:

```
SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]
```

IF - Statement:

Das `IF` Statement wird verwendet um eine Bedingung zu überprüfen und je nach dem ob die Bedingung wahr oder falsch ist, wird ein anderer Befehl ausgeführt.

3 Arten von `IF` - Statements:

- `IF - THEN`: Wenn (Argument) Dann (Befehl)
- `IF - THEN - ELSE`: Wenn (Argument) Dann (Befehl) Sonst (Befehl)
- `IF - THEN - ELSEIF - ELSE`: Wenn (Argument1) Dann (Befhel1), Wenn (Argument2) Dann (Befehl2), ..., Sonst (Befehl)

Syntax:

```
IF condition THEN
   statements;
END IF;
```

CASE - Statement:

Das `CASE` Statement wird verwendet um eine Bedingung zu überprüfen und je nach dem ob die Bedingung wahr oder falsch ist, wird ein anderer Befehl ausgeführt.

Syntax:

```
CASE case_value
   WHEN when_value1 THEN statements
   WHEN when_value2 THEN statements
   ...
   [ELSE else-statements]
END CASE;
```

LOOP - Statement:

Das `LOOP` Statement wird verwendet um eine Schleife zu erstellen.

Syntax:

```
[label]: LOOP
    ...
    -- terminate the loop
    IF condition THEN
        LEAVE [label];
    END IF;
    ...
END LOOP;
```

WHILE - LOOP:

Das `WHILE` Statement wird verwendet um eine Schleife zu erstellen, welche so lange läuft bis ein gewisses Argument erfüllt ist.

Syntax:

```
WHILE counter <= day DO
        CALL InsertCalendar(currentDate);
        SET counter = counter + 1;
        SET currentDate = DATE_ADD(currentDate ,INTERVAL 1 day);
    END WHILE;
```

REPEAT - LOOP:

Das `REPEAT` Statement wird verwendet um eine "DO - WHILE" Schleife zu erstellen:

Syntax:

```
REPEAT
        SET result = CONCAT(result,counter,',');
        SET counter = counter + 1;
    UNTIL counter >= 10
    END REPEAT;
```

SHOW WARNINGS:

Mit dem `SHOW WARNINGS` Befehl kann man sich detaillierte informationen über Warnungen anzeigen lassen

Syntax:

```
SHOW WARNINGS [LIMIT [offset,] row_count]
```

SHOW ERRORS:

Mit dem `SHOW ERROS` Befehl kann man sich detaillierte informationen über Fehler anzeigen lassen.

Syntax:

```
SHOW ERRORS [LIMIT [offset,] row_count];
```

CURSOR:

Ein `CURSOR` wird verwendet um durch die Ergebnisse einer `SELECT` Abfrage zu iterieren. Meistens wird ein `CURSUR` verwendet, wenn man einzelne Zeilen verarbeiten muss:

```
-- declare a cursor
DECLARE cursor_name CURSOR FOR
SELECT column1, column2
FROM your_table
WHERE your_condition;

-- open the cursor
OPEN cursor_name;

FETCH cursor_name INTO variable1, variable2;
-- process the data


-- close the cursor
CLOSE cursor_name;
```

Es ist empfelenswert einen `CURSOR` immer zu schliesen, wenn er nicht mehr verwendet wird:

```
CLOSE cursor_name;
```

Außerdem muss man einen `NOT FOUND HANDLER` definieren, für den Fall das der `CURSOR` keine Zeile findet:

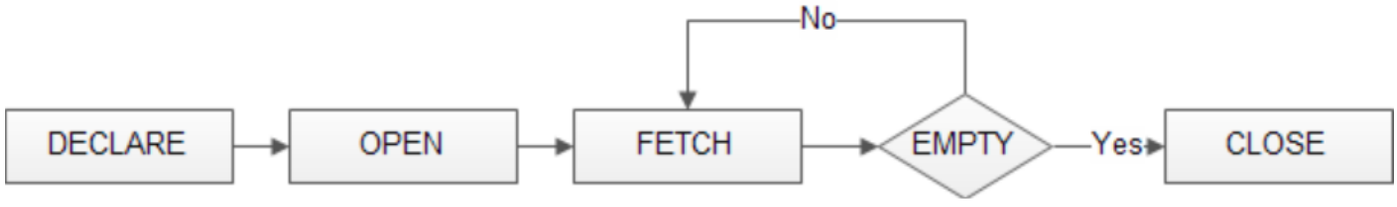
```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

Jedes mal wenn das `FETCH` Statement aufgerufen wird, versucht der `CURSOR` die nächste Zeile im Ergebniss Set zu lesen.

Die `finished` Variable zeigt an ob der `CURSOR` das Ende des Ergebnis Sets erreicht hat.

Die Handler-Deklaration muss nach den Variablen- und Cursor- Deklarationen innerhalb der `STORED PROCEDURE` passieren.

So läuft ein `CURSOR` ab:



So verwendet man einen `CURSOR` um alle Zeilen in einer `employees` Tabelle durchzulaufen und in einen String zu fügen:

```
DELIMITER $$

CREATE PROCEDURE create_email_list (
    INOUT email_list TEXT
```

```
)
BEGIN
    DECLARE done BOOL DEFAULT false;
    DECLARE email_address VARCHAR(100) DEFAULT "";

    -- declare cursor for employee email
    DECLARE cur CURSOR FOR SELECT email FROM employees;

    -- declare NOT FOUND handler
    DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET done = true;

    -- open the cursor
    OPEN cur;

    SET email_list = '';

    process_email: LOOP

        FETCH cur INTO email_address;

        IF done = true THEN
            LEAVE process_email;
        END IF;

        -- concatenate the email into the emailList
        SET email_list = CONCAT(email_address,";",email_list);
    END LOOP;

    -- close the cursor
    CLOSE cur;

END$$

DELIMITER ;
```

TRIGGERS:

`TRIGGERS` sind Datenbankobjekte, die automatisch ausgeführt werden, wenn gewisse Ereignisse wie `INSERT`, `UPDATE` oder `DELETE` ausgeführt werden.

In **MySQL** gibt es nur `TRIGGER`, welche so oft ausgeführt werden wie Reihen existieren mit auf für welche ein Ereignis ausgeführt wird. Wenn z.B: 100 Reihen gelöscht werden, wird der `TRIGGER` 100 mal ausgeführt.

Vorteile:

- Anderer Weg die Datenintegrietät zu überprüfen
- Fehler werden auf Datenbankebene gehandhabt
- Alternativer Weg Aufgaben zu erledigen. Mann muss nicht auf das Ereignis warten, da es automatisch vor oder nach einer Änderung ausgeführt wird
- nützlich um Datenänderungen in Tabellen zu überprüfen

Nachteile:

- Man kann nicht alle Validierungen nutzen. Für einfache Überprüfung kann man: `NOT NULL`, `UNIQUE`, `CHECK` und `FOREIGN KEY` verwenden.
- Aufwendige Fehlersuche, da Triggers automatisch ausgeführt werden
- (Weißt nicht wie ichs auf Deutsch übersetzen soll) Triggers may increase the overhead of the MySQL server.

Erstellen eines TRIGGERS:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger body (SQL statements)
END;
```

Löschen eines TRIGGERS:

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name;
```

Mehrere TRIGGER erstellen:

Hier wird ein `TRIGGER` ersellt, welcher vor oder nach einem existierenden `TRIGGER` als Reaktion auf das gleiche Ereignis ausgeführt wird.

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
    -- statements
END$$

DELIMITER ;
```

STORED PROCEDURE mit einem TRIGGER aufrufen:

```
DELIMITER $$

CREATE TRIGGER before_accounts_update
BEFORE UPDATE
ON accounts FOR EACH ROW
```

```
BEGIN
    CALL CheckWithdrawal (
        OLD.accountId,
        OLD.amount - NEW.amount
    );
END$$

DELIMITER ;
```

SHOW TRIGGERS:

Mit `SHOW TRIGGERS` kann man sich alle definierten Trigger anzeigen lassen:

```
SHOW TRIGGERS
[{FROM | IN} database_name]
[LIKE 'pattern' | WHERE search_condition];
```

VIEWS

`VIEWS` sind virtuelle Tabellen, welche das Ergebniss einer `SELECT` repräsentieren. Sie existieren nicht als physische Speicherung von Daten, sondern als gespeicherte Abfrage.

VIEW anlegen:

```
CREATE [OR REPLACE] VIEW [db_name.]view_name [(column_list)]
AS
  select-statement;
```

VIEW aufrufen:

```
SELECT * FROM customerPayments;
```

VIEW löschen:

```
DROP VIEW [IF EXISTS] view_name;
```

Vorteile:

- Vereinfachen komplizierter Abfragen
- Alle Logischen Zugriffe konsistent halten
- Sicherer (Man kann festlegen auf welche Daten Nutzer zugreifen können)
- Abwärtskompatibilität (Wenn z.B eine Tabelle in mehrere kleine Zerlegt wird, kann man eine View erstellen, auf welche dann ältere Geräte zugreifen können, als währe es die alte Tabelle)