FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Databases
SQL Practical Course

Prof. Dr. rer. nat. Alexander Voß
Inform-Professur

FH Aachen
University of applied Sciences

V2024/25

0x01
Selection
Projection

0x02
Joins

0x03
Group
Functions

0x04
Subselects

0x00
General
Information

0x05
Schemas
Tables

0x08
Stored
Procedures

0x07
Data
Management

0x06
Time/Data
View
Transaction

Unit 0x00
General Information

A few notes in advance

In this course, the text refers to **attributes, columns, tables, keys, schemas** or **databases**, just to name a few. Here is a brief definition of the terms.

- **Attributes, tables:** The table on the right named shop_product models products from the AMI_ZONE company.
  The columns of the table are the so-called attributes, i.e. the properties of each product.
  Each row is an entity/object - here a product.

	name	unit	price
1	Spinach	PK	1.99
2	Four Cheese Pizza	PC	2.39
3	Spinach Pizza	PC	2.29
4	Fish Fingers	PK	1.99
5	Pasta Pan	PK	3.29
6	Carrots	KG	0.39

table shop_product

- **Data types:** Attributes are typed. Standard types in **MariaDB** are, for example
  - **INT:** classic integer
  - **FLOAT(M,D):** floating point, (M,D) means M total digits, D decimal places
  - **VARCHAR(M):** text with a maximum length of M characters
  - **DATETIME:** Time in the form YYYY-MM-DD hh:mm:ss.

A few notes in advance

- **Key, id:** An entity almost always has some kind of unique identifier or id, also called a key. This identifies the specific entity and is unique.
- **Foreign key:** An attribute in one table that represents a key in another table is called a foreign key. The exact definition and meaning follows.
- **Schema/Database:** Tables are grouped together in a schema. Depending on the DBMS, it may simply be called a database instead of a schema.
- **Indexes:** Additional data structures, e.g. B-trees, are created automatically or intentionally to allow quick access to data.

id	product	category	price
4	Fish Fingers	1	1.99
5	Pasta Pan	1	3.29
6	Carrots	2	0.39
7	Onions	2	0.29
8	Bananas	2	1.29
9	Garlic	2	0.98
10	Pork Sausage	3	1.99
11	Meatballs	3	2.35
12	Milk	4	0.79
13	Quark	4	0.99
14	Yoghurt	4	0.98
15	Cola light	5	1.50

category	name
1	Frozen Goods
2	Fruits, Vegetables
3	Sausages, Cold Meat
4	Milk Products
5	Cold Drinks
6	Barbecue Products
7	Snacks
8	Hot Drinks
9	Convenience Foods
10	Baked Goods
11	Magazines
12	Services

Related product data from the ami_zone database

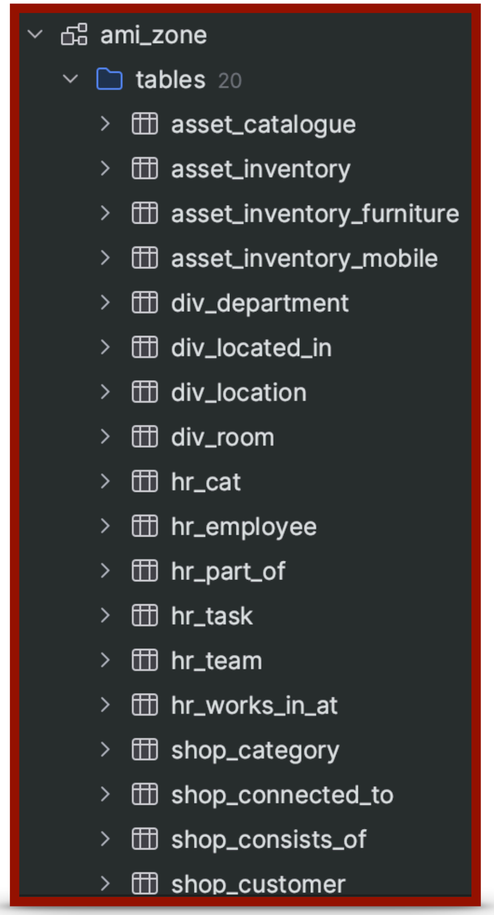
A few notes in advance

- **Upper/lower case:** Whether commands, attributes and table names can or must be written in upper or lower case depends on the DBMS, its settings and the underlying operating system. It is not relevant to the exams.
  Some commands are case-insensitive, some are not. You need to check the documentation.
- **NULL:** In addition to the typical range of values for each data type, there is another value that an attribute can have: NULL. The meaning of the value NULL is given by the context, often the value is simply used for "no specification" or "not yet specified". The column definition of a table may allow or disallow NULL, but if the value is allowed, it often requires special handling in queries.
- **Script:** The SQL commands for a unit are also available as an sql file, so you can try them out instead of typing them in.

Sample Data AMI_ZONE

The Idea

- AMI_ZONE is a fictitious company with individual departments and employees at different locations that delivers everyday necessities when there is no time to go shopping.
  Typical examples are frozen pizzas or the latest c't magazine.
- The database contains a lot of data and relationships and is used to practise SQL commands.

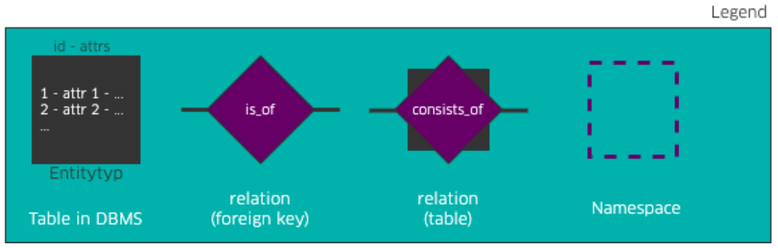


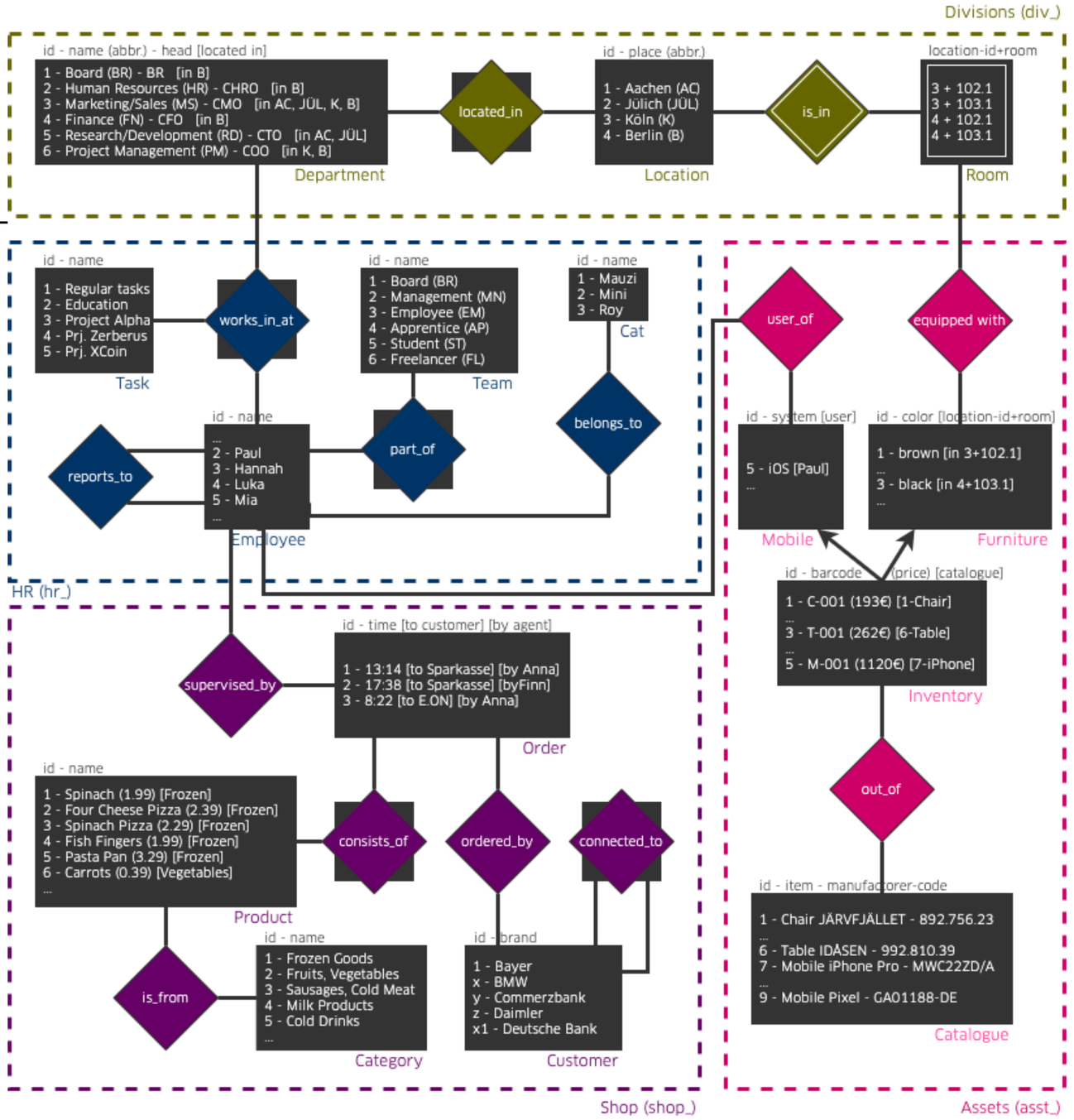
database AMI_ZONE

Sample Data AMI_ZONE

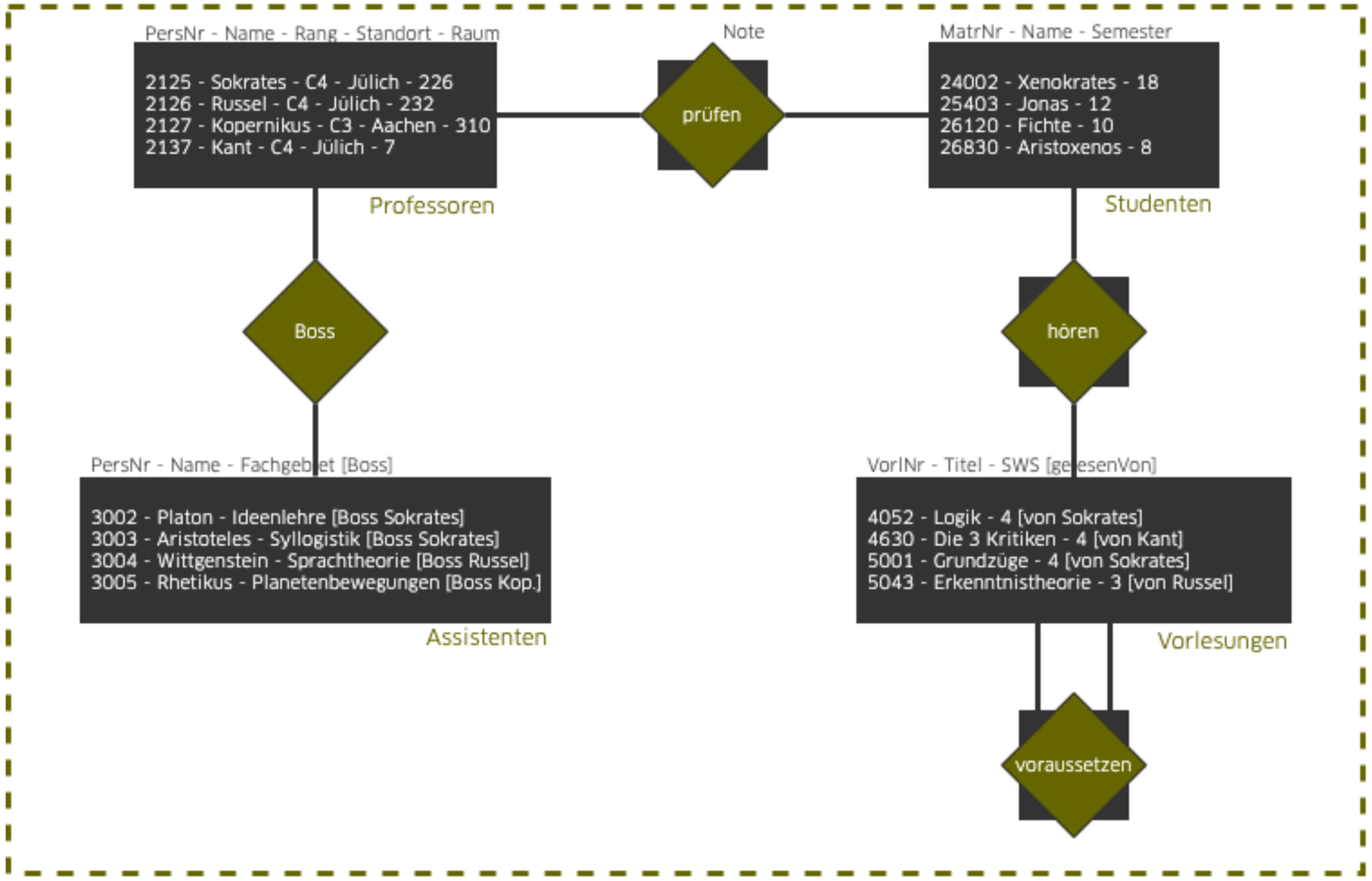
Modelled fields

- Departments/ Locations (div)
- Orders/ Products (shop)
- Employees/ Supervisors (hr)
- Equipment/ Resources (assets)
- and some more.



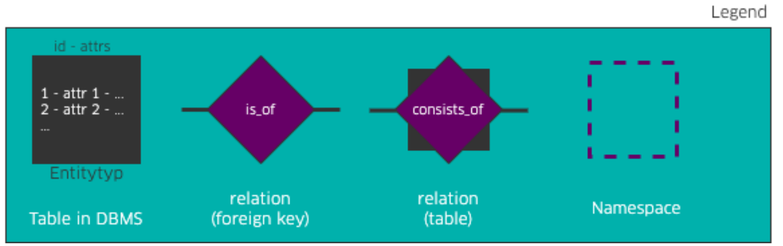


Sample Data AMI_KEMPER, used in exams



Modelled fields (in german)

- University staff (professors, assistants, students)
- Lectures



Unit 0x01
Selection & Projection

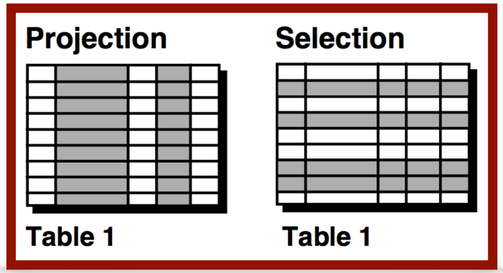
select

Goal

- Formulate data requests using the select command.

Projection and Selection

- **Projection** selects columns by specifying table attributes.
- **Selection** filters rows by specifying a criterion that must be met for each row.



from Oracle Documentation

```
SELECT
    [ALL | DISTINCT | DISTINCTROW]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [ FROM table_references
      [WHERE where_condition]
      [GROUP BY {col_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]
      [HAVING where_condition]
      [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
      [LIMIT {[offset,] row_count | row_count OFFSET offset  [ROWS EXAMINED rows_limit] } |
        [OFFSET start { ROW | ROWS }]
        [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES }] ]
      procedure|[PROCEDURE procedure_name(argument_list)]
      [INTO OUTFILE 'file_name' [CHARACTER SET charset_name] [export_options] |
        INTO DUMPFILE 'file_name' | INTO var_name [, var_name] ]
      [FOR UPDATE lock_option | LOCK IN SHARE MODE lock_option]

export_options:
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]

lock_option:
    [WAIT n | NOWAIT | SKIP LOCKED]
```

from MariaDB Documentation

use

Intention

- Define a default schema (or database, depending on the DBMS) with **use**.

Remarks

- If a default schema is specified with **use**, the explicit specification of the schema can be omitted for tables in SQL expressions.
- The explanation of the **select** command follows immediately (in short: Query all data), but with the previous **use**, the table product can be addressed here even without specifying **ami_zone**.

```sql
-- default schema
USE ami_zone;

-- all data
SELECT * FROM ami_zone.shop_product;
SELECT * FROM shop_product;
```

db_unit_0x01_training

- It is not available in all DBMS, e.g. PostgreSQL knows a default database but not **use**.

select – */From

Intention

- Query **all** records (no selection) and **all** attributes (no projection) of a table.

Remarks

- * selects all attributes in a table.
- The column headers are the attribute names or columns of the table.
- The order of the records is not specified here.

```
-- all columns and all rows
SELECT * FROM shop_product;
```

all columns and all rows

40 rows | Tx: Auto | DDL | CSV

id	name	category_id	unit	price	tax
1	Spinach	1	PK	1.99	0.07
2	Four Cheese Pizza	1	PC	2.39	0.07
3	Spinach Pizza	1	PC	2.29	0.07
4	Fish Fingers	1	PK	1.99	0.07
5	Pasta Pan	1	PK	3.29	0.07
6	Carrots	2	KG	0.39	0.07

db_unit_0x01_training (tax is now vat (value added tax=USt.)

select – Table Alias

Intention

- Provide tables in select with an alias.

Remarks

- Tables can be given an alias name, here P, which is used instead.
- This not only makes more complex expressions shorter, but also removes any ambiguity.
- Attributes then get a P. prefix, see also following slides.

```
-- tables with alias
SELECT P.* FROM shop_product P;
```

tables with alias

40 rows | Tx: Auto | DDL | CSV

id	name	category_id	unit	price	tax
1	Spinach	1	PK	1.99	0.07
2	Four Cheese Pizza	1	PC	2.39	0.07
3	Spinach Pizza	1	PC	2.29	0.07
4	Fish Fingers	1	PK	1.99	0.07
5	Pasta Pan	1	PK	3.29	0.07
6	Carrots	2	KG	0.39	0.07

db_unit_0x01_training

select – Projection and Column Alias

Intention

- Query only **certain** attributes (projection).

Remarks

- Column names, e.g. ID, can be specified explicitly.
- AS is optional.
- The use of quotes is complicated. MariaDB allows both single and double quotes for string literals. In PostgreSQL, for example, double quotes identify objects within the database.

```
-- note column-names
SELECT
  id AS 'ID',
  name 'Product',
  price 'Price [€]'
FROM shop_product;
```

note column-names

40 rows

ID	Product	`Price [€]`
1	Spinach	1.99
2	Four Cheese Pizza	2.39
3	Spinach Pizza	2.29
4	Fish Fingers	1.99
5	Pasta Pan	3.29
6	Carrots	0.39

db_unit_0x01_training

select – Functions

Intention

- Use arithmetic expressions and functions (e.g. round, concat).

Remarks

- Consulting the documentation is the best way to find out about the calling specification and possible variations of similar functions.

```
31  -- calculated values
32  SELECT
33      id 'ID ',
34      concat(name,' [',unit,']') 'Product',
35      price 'Price [€]',
36      round(price/(1.0+tax),2) 'excl. Tax',
37      round(price/(1.0+tax)*tax,2) 'Tax '
38  FROM shop_product;
```

Output | calculated values

40 rows | CSV

	`ID `	Product	`Price [€]`	`excl. Tax`	`Tax `
1	1	Spinach [PK]	1.99	1.86	0.13
2	2	Four Cheese Pizza [PC]	2.39	2.23	0.16
3	3	Spinach Pizza [PC]	2.29	2.14	0.15
4	4	Fish Fingers [PK]	1.99	1.86	0.13
5	5	Pasta Pan [PK]	3.29	3.07	0.22
6	6	Carrots [KG]	0.39	0.36	0.03

db_unit_0x01_training

select – Operators and NULL

Intention

- Use arithmetic expressions such as is NULL and functions with NULL.

Remarks

- Caution: Expressions are NULL if one of the operands is NULL.
- coalesce returns the first non-NULL parameter.

```
43 ✔  SELECT
44        brand,
45        risk_of_loss_percent,
46        risk_of_loss_percent is NULL 'NULL?',
47       coalesce(100-risk_of_loss_percent,100.00) 'Fulfilment'
48    FROM shop_customer;
```

Output | calculated values involving NULL

12 rows

	brand	risk_of_loss_percent	`NULL?`	Fulfilment
7	E.ON	<null>	1	100.00
8	Infinion	<null>	1	100.00
9	Lufthansa	<null>	1	100.00
10	RWE	2.00	0	98.00
11	SAP	<null>	1	100.00
12	Sparkasse	<null>	1	100.00

db_unit_0x01_training

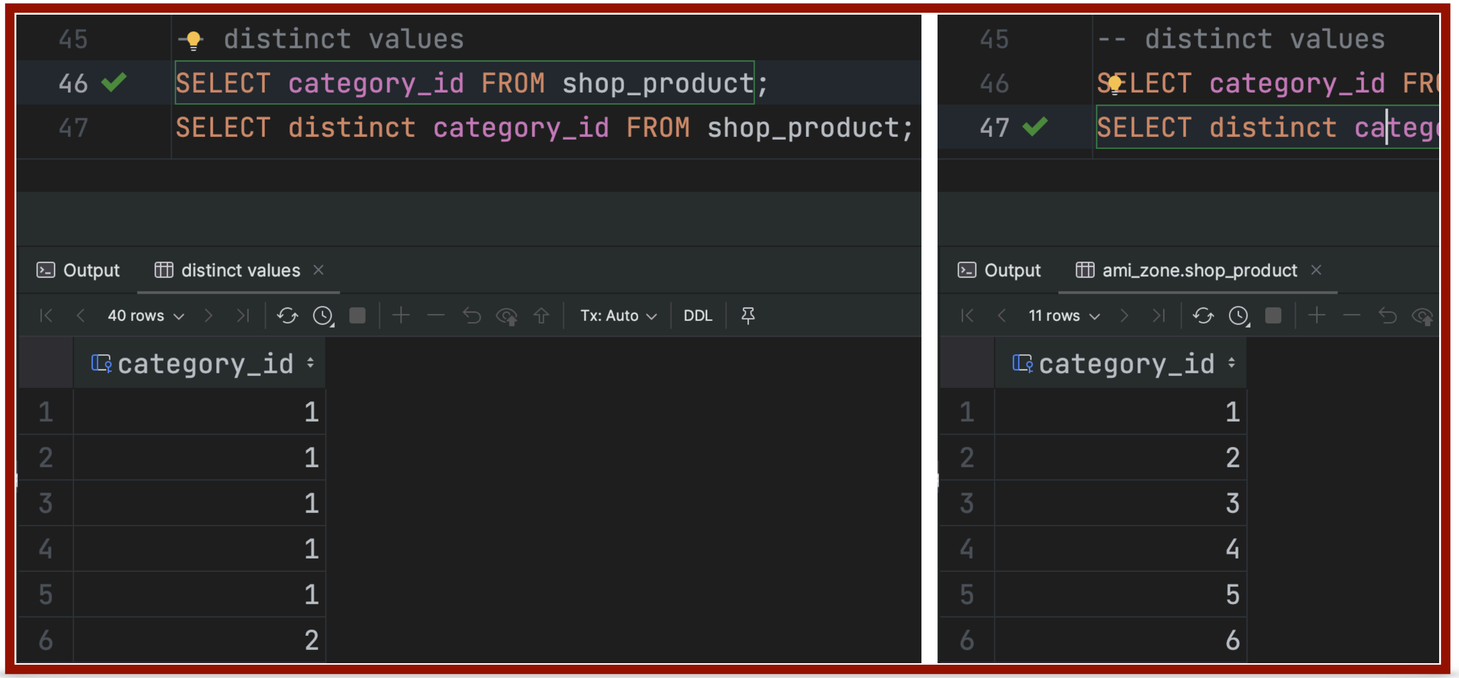
select – distinct

Intention

- Duplicate data should not be included.

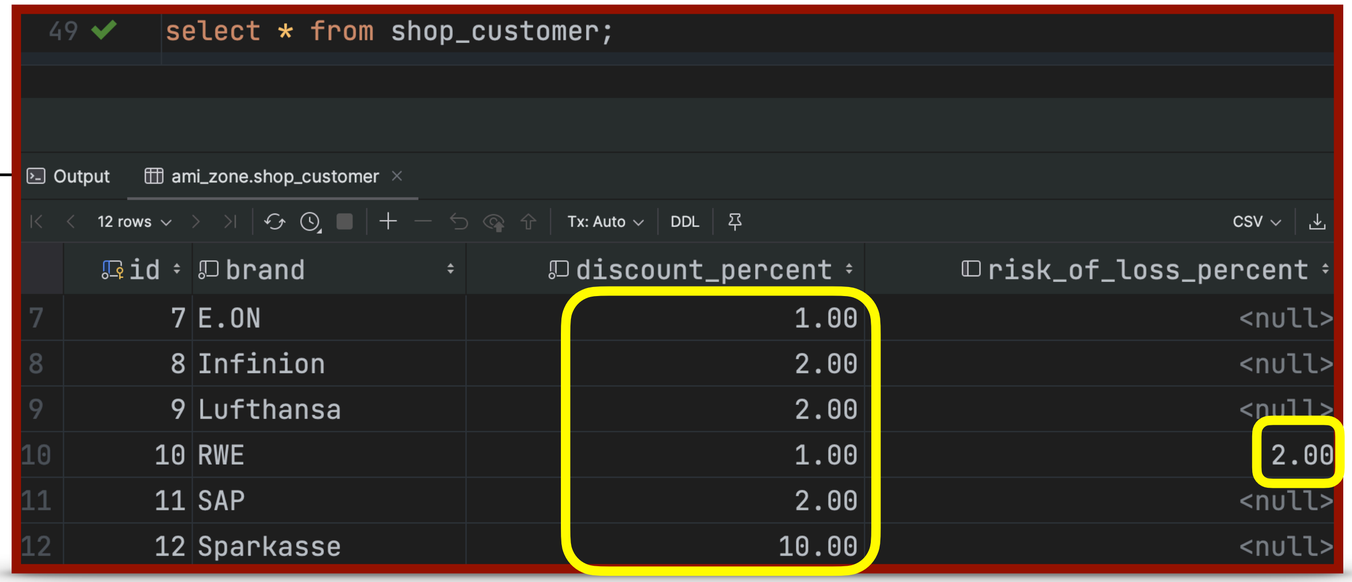
Remarks

- Without distinct results may occur more than once.
- This is where the analogy with a mathematical set breaks down.



db_unit_0x01_training

select – count

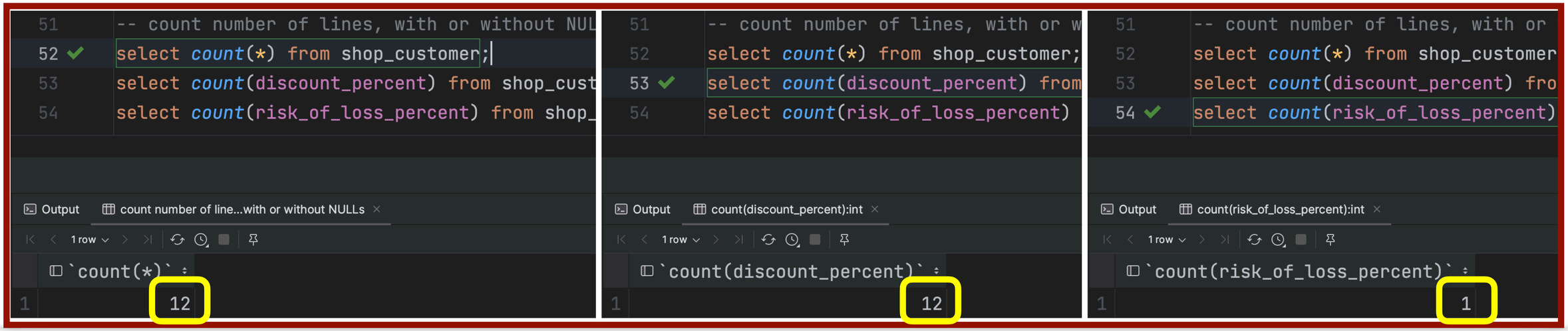


Intention

- Determine the number of records.

Remarks

- A * in `count` causes all records to be counted, whereas specifying an attribute causes only records other than `NULL` to be counted.



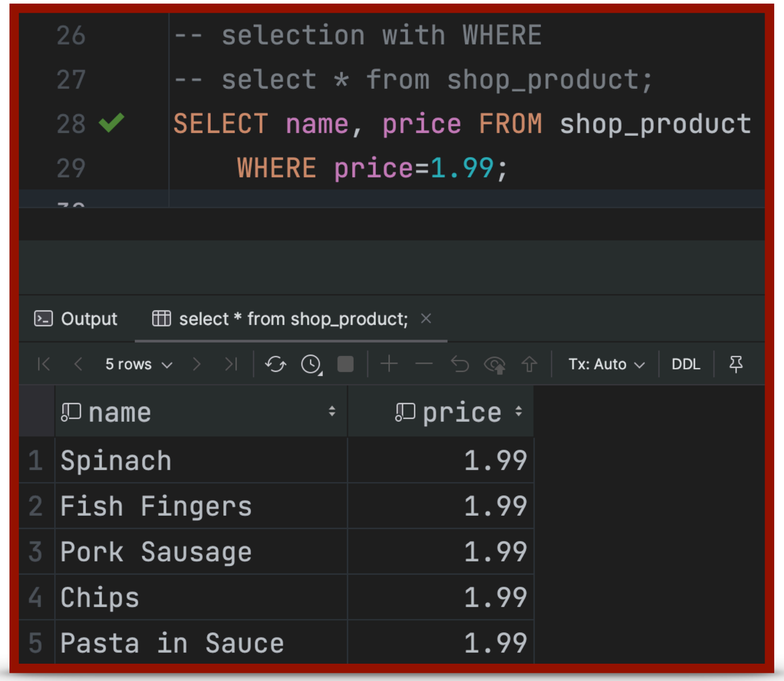
select – where

Intention

- Query only table attributes or expressions matching a condition (after `where`).

Remarks

- The condition, here `price=1.99`, tests for equality, but of course the caveats about comparing floating point numbers with regard to rounding errors still apply.



db_unit_0x01_training

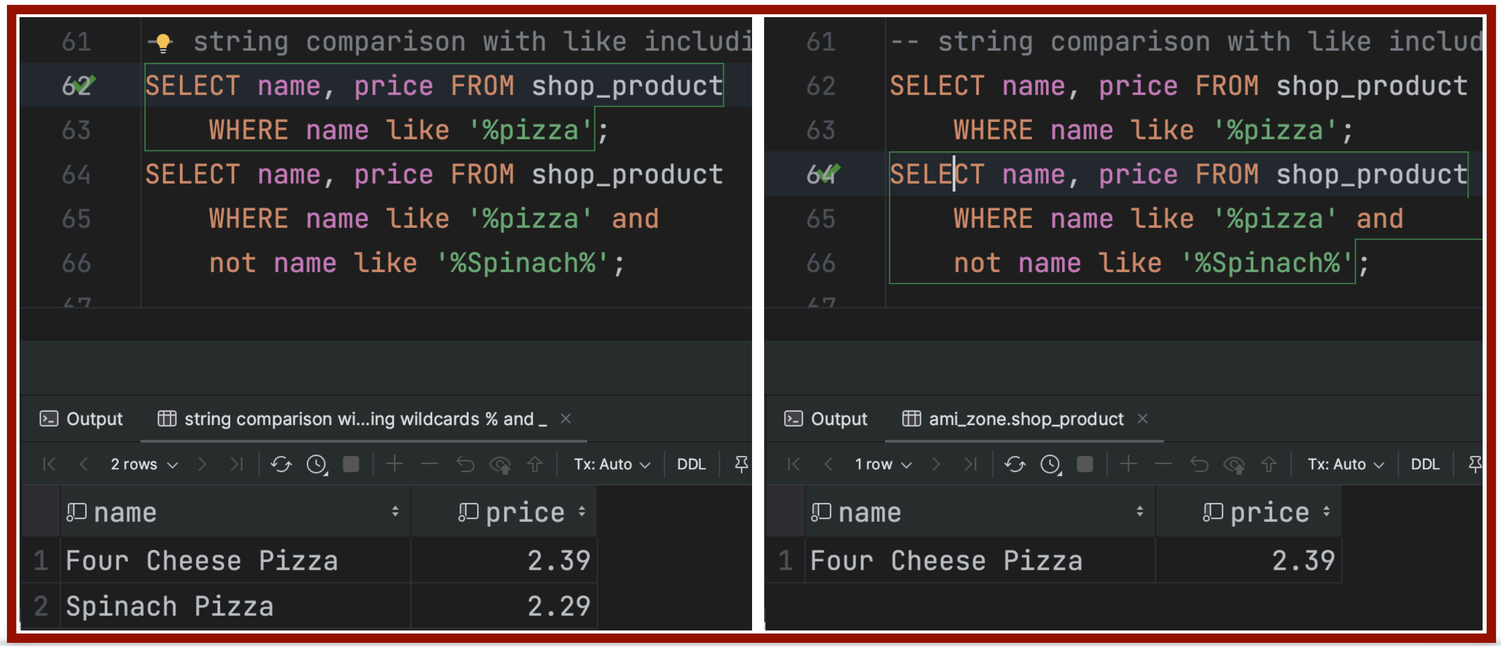
select – Operators

Intention

- Use operators such as like, not, and in where conditions.

Remarks

- like is used for string comparison, but the operation may or may not be case-sensitive, depending on the DBMS.
- The % wildcard allows any character (including none),
- The _ wildcard allows exactly one character.



db_unit_0x01_training

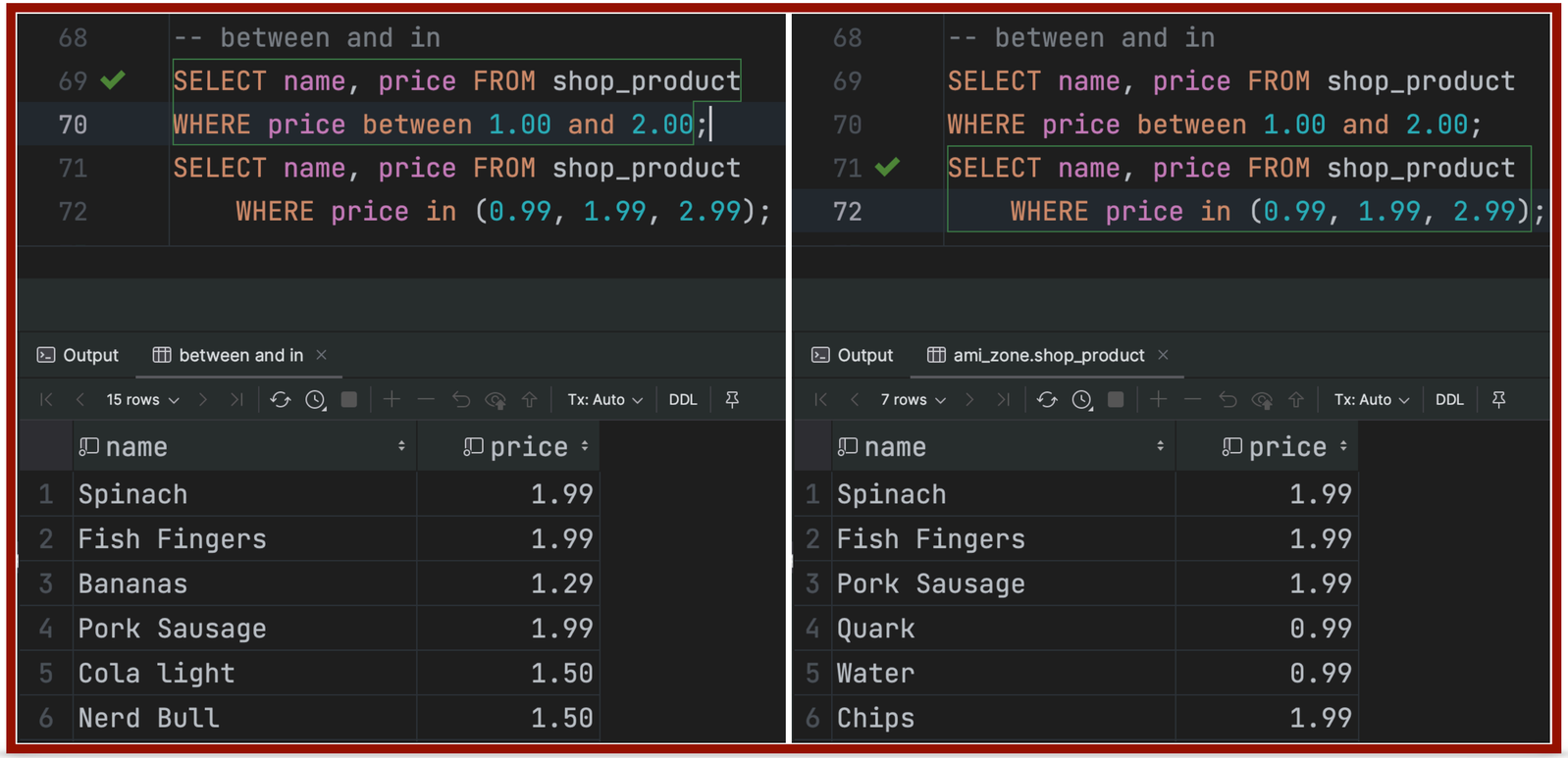
select – Operators

Intention

- Use operators such as between...and, in in where conditions.

Remarks

- The boundaries at between...and are inclusive.



db_unit_0x01_training

select – case

Intention

- The case statement allows an output per case.

```
SELECT name, price,
       case
           when price <= 2.00 then 'cheap'
           when price <= 5.00 then 'ok'
           else 'expensive'
        end as 'x'
    FROM shop_product
    WHERE price between 1.00 and 10.00;
```

db_unit_0x01_training

Brown Bread	1.90	cheap
Kicker	2.00	cheap
c't	3.70	ok
Computerbild	2.20	ok
Zeit	4.20	ok
GameStar	6.50	expensive

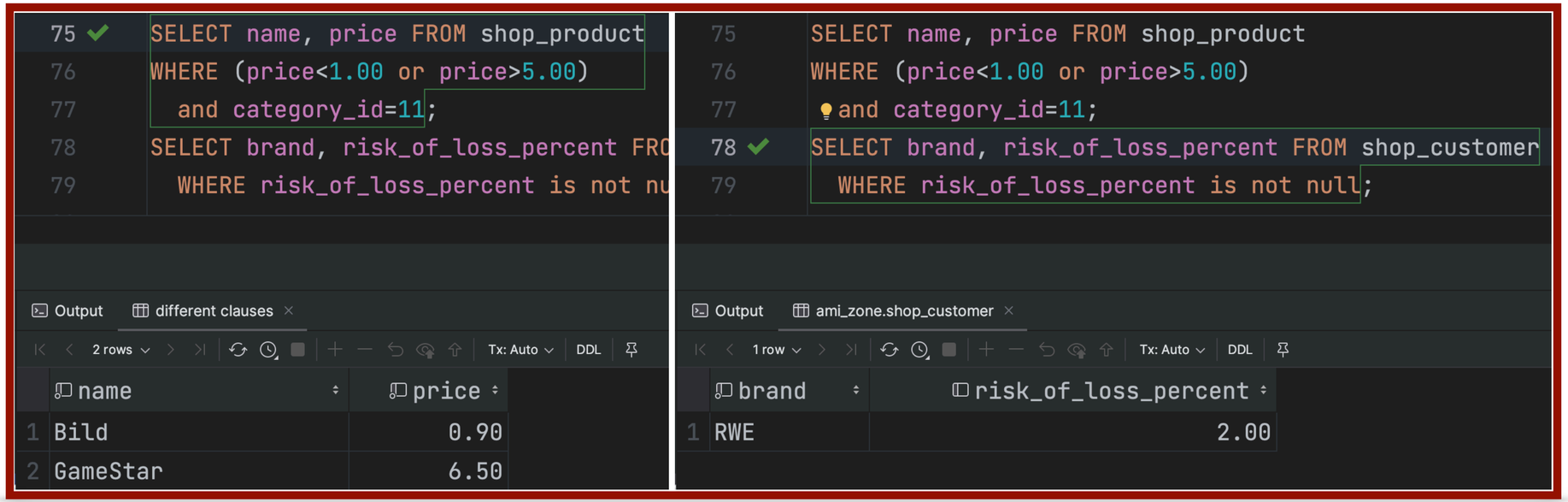
select – Operators and NULL

Intention

- Use operators such as is NULL or is not NULL and parentheses in where conditions.

Remarks

- As before, be careful with NULL values. The result of an arithmetic operation with NULL is also NULL.



db_unit_0x01_training

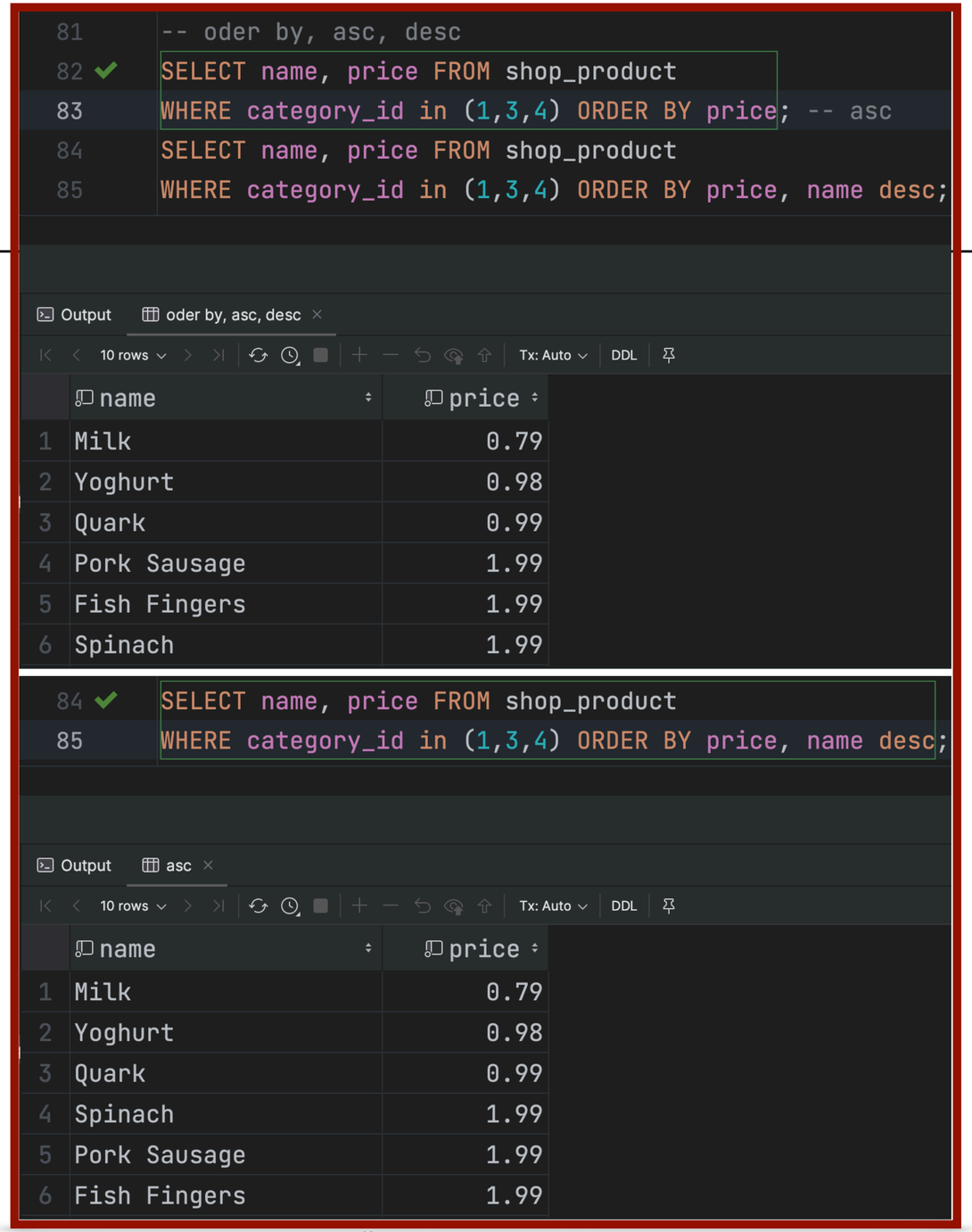
select – Sorting

Intention

- Sort the found results by criteria (order by), ascending (asc), or descending (desc).
- Sort by multiple criteria.

Remarks

- The lower result set sorts the 1.99-items alphabetically, descending by command.



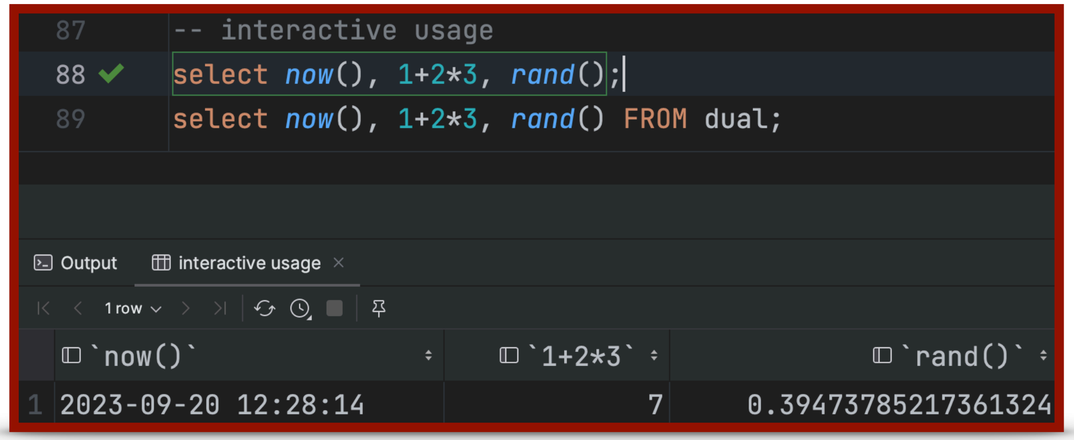
select – Interactive

Intention

- Interactive use of **select**.

Remarks

- In some DBMS, such as Oracle, a table must always be specified in the **select** command. This is then the (pseudo) table **dual**.



db_unit_0x01_training

Check

Tasks

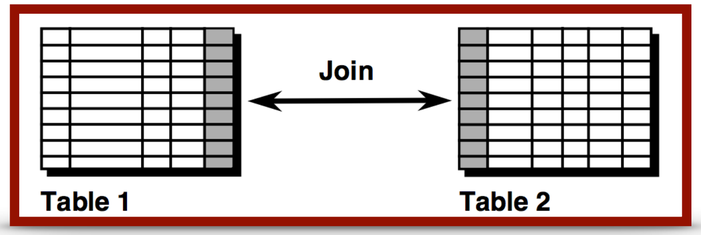
1. Print the entire contents of each of the tables `div_department`, `shop_customer`, `hr_employee`, `shop_product`, `hr_team` and `shop_category` from `AMI_ZONE`.
2. Find out how many different `VAT` rates there are in the `shop_product` table and print them.
3. Which customers are a `Bank` or `Sparkasse` by name? Identify them and print only the name and discount.
4. Find the monthly salary (assume 12 months) of all employees whose manager is `Mia` or `Ben`. Give the column the heading `Monthly salary` and round the monthly salary to full Euros. (Look up the `id` of `Mia` and `Ben`, no sub-query required yet).
5. How many employee names begin with an `M`? And how many contain an `M`? Find out the number of each.

Unit 0x02
JOINS

Joins

Goal

- Formulate data queries across multiple tables using **joins**.
- Different types of joins
  - Inner,
  - Left Outer,
  - Right Outer,
  - Full Outer,
  - Cross,
  - Self and
  - Natural Joins.
- Joins are performed using the `select` command.



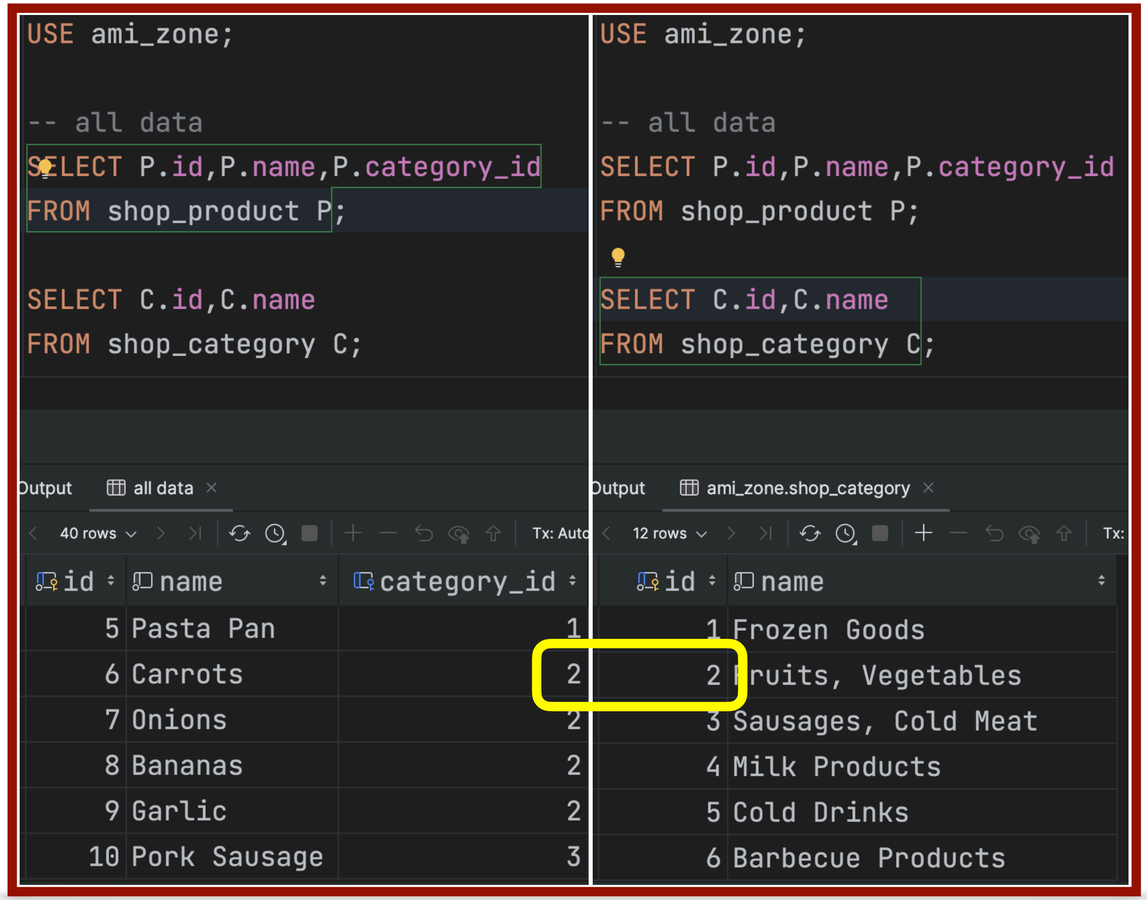
from Oracle Documentation

Motivation

Relationship product–category

- Each product is assigned to a category (N:1) and this relationship is realised by the foreign key category_id in shop_product.
- It would be nice if we could obtain the **category itself** instead of the category_id:

id	P.name	C.name
5	Pasta Pan	Frozen Goods
6	Carrots	Fruits, Vegetables
7	Onions	Fruits, Vegetables
8	Bananas	Fruits, Vegetables
9	Garlic	Fruits, Vegetables
10	Pork Sausage	Sausages, Cold Meat



db_unit_0x02_training

Motivation

Relationship product–category

- **First approach:** If we specify both tables after `from` in the `select`, we get the Cartesian product - in short, every product with every category (a so-called implicit **cross join**).
- How can we cleverly select so that the result contains only **matching** rows?
  In other words: What is the concrete `where` condition?

```
# motivation: cross
SELECT P.id,P.name,P.category_id,C.id,C.name
FROM shop_product P, shop_category C;
```

utput # motivation: cross

480 rows CSV

P.id	P.name	category_id	C.id	C.name
1	Spinach	1	10	Baked Goods
1	Spinach	1	11	Magazines
1	Spinach	1	12	Services
2	Four Cheese Pizza	1	1	Frozen Goods
2	Four Cheese Pizza	1	2	Fruits, Vegetables
2	Four Cheese Pizza	1	3	Sausages, Cold Meat

db_unit_0x02_training

Motivation

Relationship product–category

- We are only interested in the rows where the foreign key category_id matches the category id - this obviously results in the correct pairing and is explicitly expressed by the where condition (called an **inner join**).
- The DBMS recognises this join and does not first determine all the combinations, only to filter out almost all of them again…

```
SELECT P.id,P.name,P.category_id,C.id,C.name
FROM shop_product P, shop_category C
WHERE P.category_id=C.id;
```

Output # motivation: explicit join

40 rows

P.id	P.name	category_id	C.id	C.name
3	Spinach Pizza	1	1	Frozen Goods
4	Fish Fingers	1	1	Frozen Goods
5	Pasta Pan	1	1	Frozen Goods
6	Carrots	2	2	Fruits, Vegetables
7	Onions	2	2	Fruits, Vegetables
8	Bananas	2	2	Fruits, Vegetables

db_unit_0x02_training

Motivation

Relationship product-category

- This join variant uses an explicit `join` command and the `on` condition is identical to the `where` condition before.
- And now for the recap.

```
SELECT P.id,P.name,C.name
FROM shop_product P
JOIN shop_category C ON P.category_id=C.id;
```

notivation: join by command | ami_zone.shop_category | goal | motivation

40 rows

id	P.name	C.name
3	Spinach Pizza	Frozen Goods
4	Fish Fingers	Frozen Goods
5	Pasta Pan	Frozen Goods
6	Carrots	Fruits, Vegetables
7	Onions	Fruits, Vegetables
8	Bananas	Fruits, Vegetables

db_unit_0x02_training

Cross Join

Intention

- Specifying multiple tables with no further restrictions results in the cross product of the entries - an implicit **cross join** (first select). That is, each entity in the first table is combined with each entity in the second table (similarly for multiple tables).
- An explicit cross join (second select) is equivalent.

```
SELECT P.id, P.name, P.category_id, C.id, C.name
FROM shop_product P,shop_category C;

SELECT P.id, P.name, P.category_id, C.id, C.name
FROM shop_product P CROSS JOIN shop_category C;
```

tput Result 3

480 rows CSV

P.id	P.name	category_id	C.id	C.name
1	Spinach	1	10	Baked Goods
1	Spinach	1	11	Magazines
1	Spinach	1	12	Services
2	Four Cheese Pizza	1	1	Frozen Goods
2	Four Cheese Pizza	1	2	Fruits, Vegetables
2	Four Cheese Pizza	1	3	Sausages, Cold Meat

db_unit_0x02_training

Inner Join

Intention

- As described above, you are often only interested in data that 'belongs together', such as the category of a product.
- An inner join is defined as a selection on the cross-product. Both joins are equivalent here, but the upper form is considered obsolete.

Remarks

- The explicit `inner join` expects an `on` specifying which data belongs together. `inner` is optional here.

```
-- inner join (old, but this is the formal definition)
SELECT P.id, P.name, P.category_id, C.id, C.name
FROM shop_product P,shop_category C
WHERE P.category_id=C.id;
-- inner join, use this
SELECT P.id, P.name, P.category_id, C.id, C.name
FROM shop_product P INNER JOIN shop_category C
ON P.category_id=C.id;
```

tput | inner join, use this

40 rows | CSV

P.id	P.name	category_id	C.id	C.name
10	Pork Sausage	3	3	Sausages, Cold Meat
11	Meatballs	3	3	Sausages, Cold Meat
12	Milk	4	4	Milk Products
13	Quark	4	4	Milk Products
14	Yoghurt	4	4	Milk Products
15	Cola light	5	5	Cold Drinks

db_unit_0x02_training

Inner Join and Self Join

Example

- All customers who have a purchase order - and only them!
- All employees together with the manager.

Remarks

- The latter is a self-relationship, a so-called **self-join**.

```
SELECT O.id, O.customer_id, C.id, C.brand
    FROM shop_order O
    INNER JOIN shop_customer C ON O.customer_id=C.id;
```

tput # example 1

3 rows

O.id	customer_id	C.id	brand
1	12	12	Sparkasse
2	12	12	Sparkasse
3	7	7	E.ON

db_unit_0x02_training

```
SELECT E.id,E.name,E.employee_id,EE.id,EE.name "Boss"
    FROM hr_employee E
    INNER JOIN hr_employee EE on E.employee_id=EE.id;
```

tput example 2

23 rows

E.id	name	employee_id	EE.id	Boss
5	Mia	1	1	Board
6	Ben	5	5	Mia
7	Emma	5	5	Mia
8	Sofia	5	5	Mia
9	Jonas	5	5	Mia
10	Anna	6	6	Ben

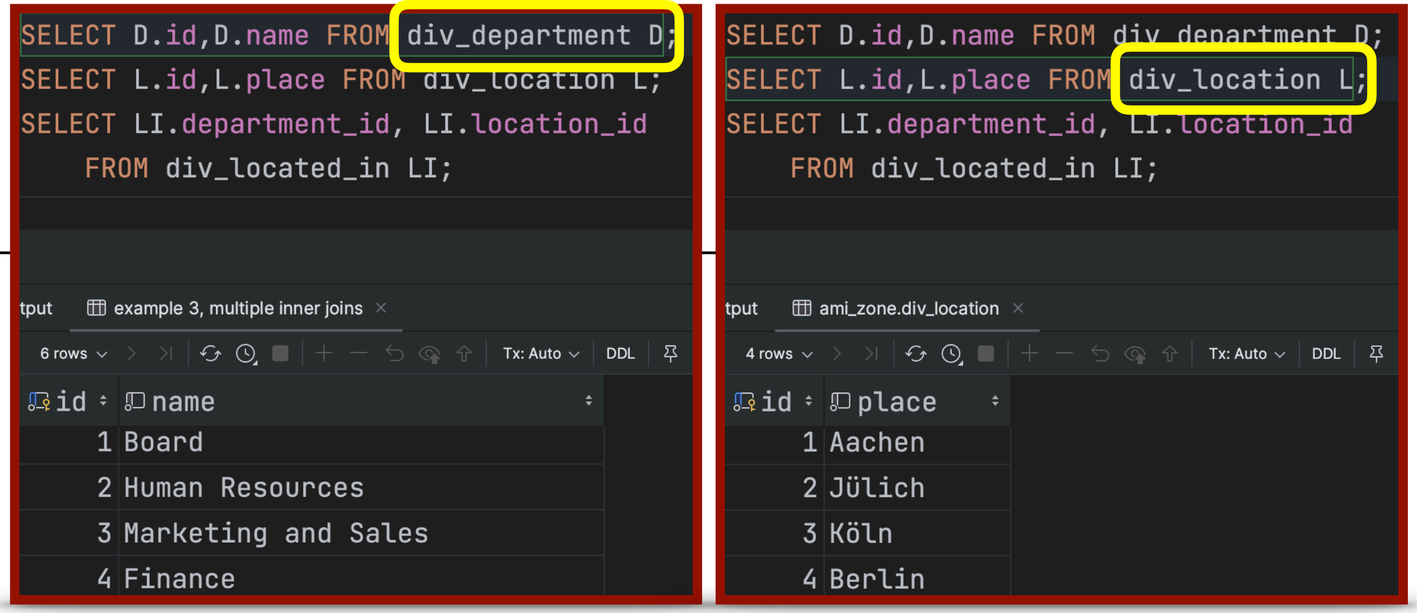
Inner Join, Multiple Tables

Example

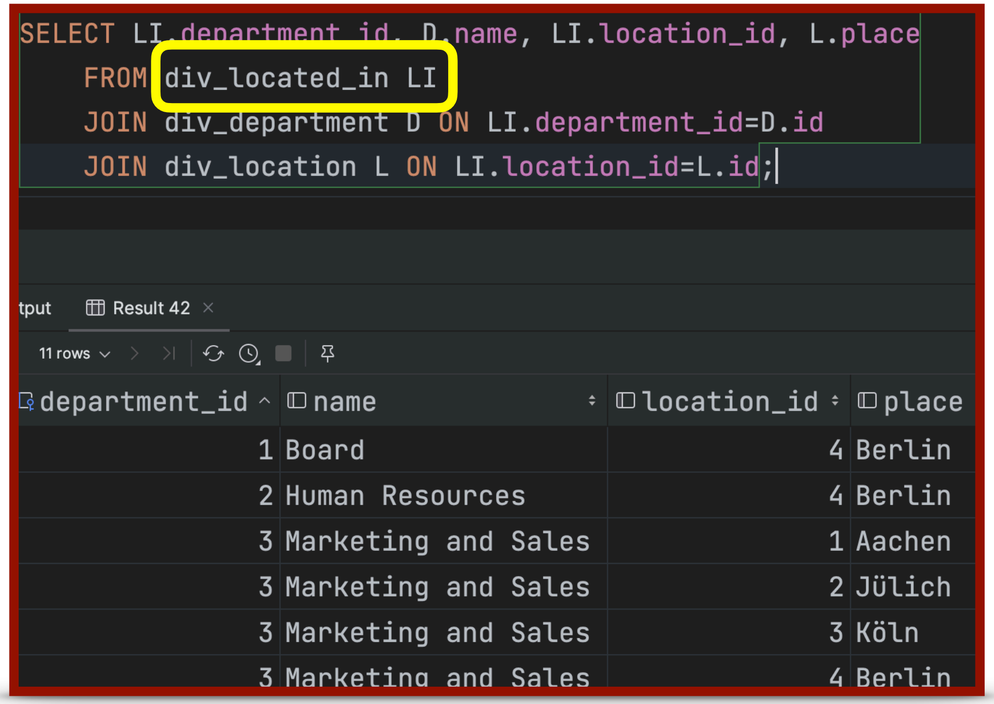
- Departments can be spread across multiple locations and locations can host multiple departments.
- This is an N:M relation.

Remarks

- Multiple joins can be easily combined. The definition remains the same.



db_unit_0x02_training



Inner Join

Example

- Here you can see who works in which department, what they work on and how many hours they work each week.

Remarks

- Although this is a relational table, the entities have an attribute, the hours.
  They depend on the employee, the department and also on the task.

```
SELECT E.name,D.name,T.name,W.hours_per_week
    FROM hr_works_in_at W
    JOIN hr_employee E on E.id = W.employee_id
    JOIN div_department D on D.id = W.department_id
    JOIN hr_task T on T.id = W.task_id;
```

utput Result 46

32 rows CSV

E.name	D.name	T.name	hours_per_week
Tim	Marketing and Sales	Regular tasks	40.00
Max	Research and Development	Education	20.00
Leonie	Research and Development	Education	20.00
Finn	Research and Development	Project Alpha	10.00
Leon	Project Management	Project Zerberus	10.00
Lilly	Research and Development	Project Zerberus	10.00

db_unit_0x02_training

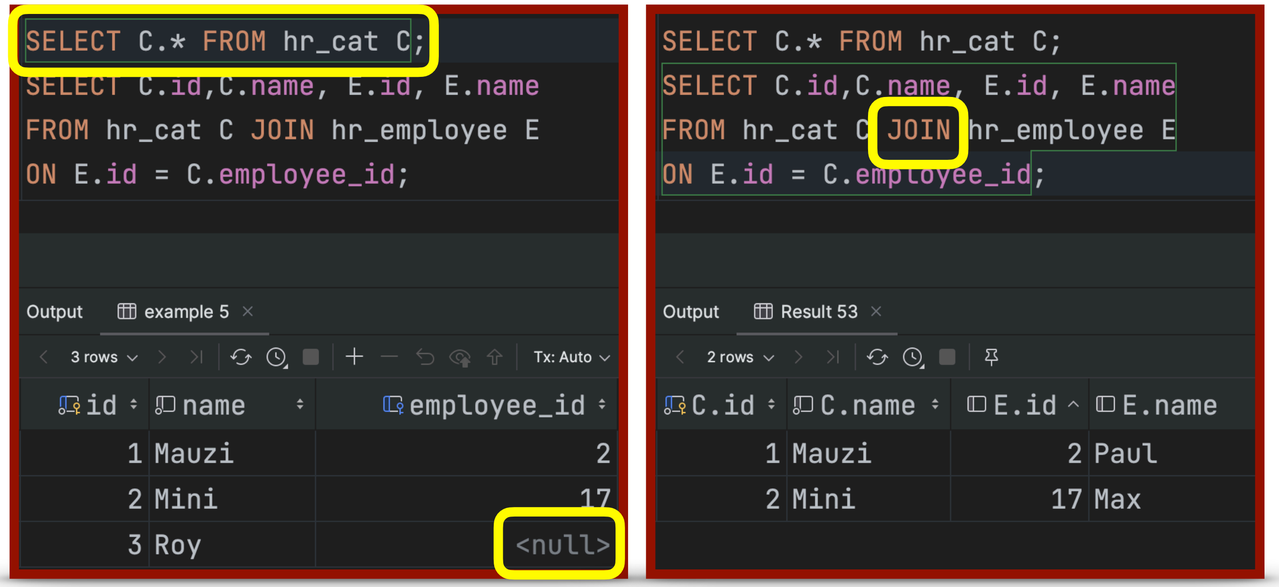
Inner Join, Missing Counterpart

Example

- Here, all cats are associated with the people who care for them.

Remarks

- Note that the entities for which there is no counterpart (i.e. containing a NULL in the foreign key) are dropped (here Roy).
  This is a fundamental property of the inner join.



db_unit_0x02_training

Outer Join

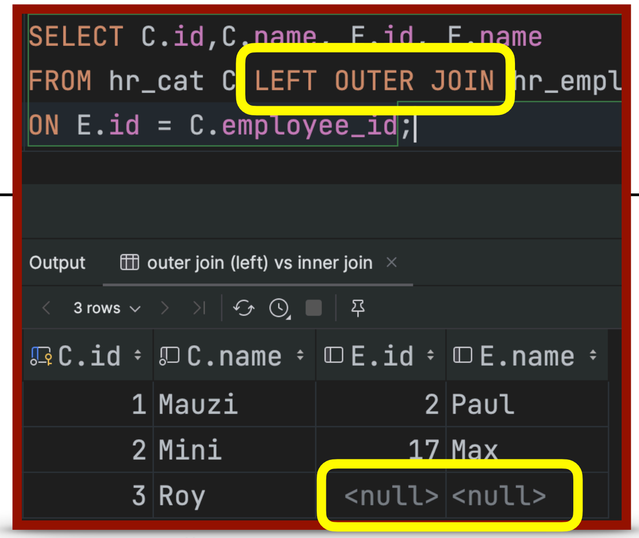
Intention

- **Query 'related' data and data for which there is no counterpart.**
- Because there are two sides to the relationship, you also need to specify in the outer join which side you want to have completely in the result set.
  There are **left outer join**, **right outer join** and **full outer join**:
  - A **left outer join** considers all entities in the entity set **on the left of the relationship**, even if there are no related entities in the entity set on the right. These attributes are then `NULL`.
  - The **right outer join** is similar, but with the roles reversed. This means that it considers all entities on the right side, and fills in the attributes of the 'related' entity on the left, or `NULL`.
  - The **full outer join** is the **union** of the left outer join and the right outer join. This means that all entities on both sides are included.

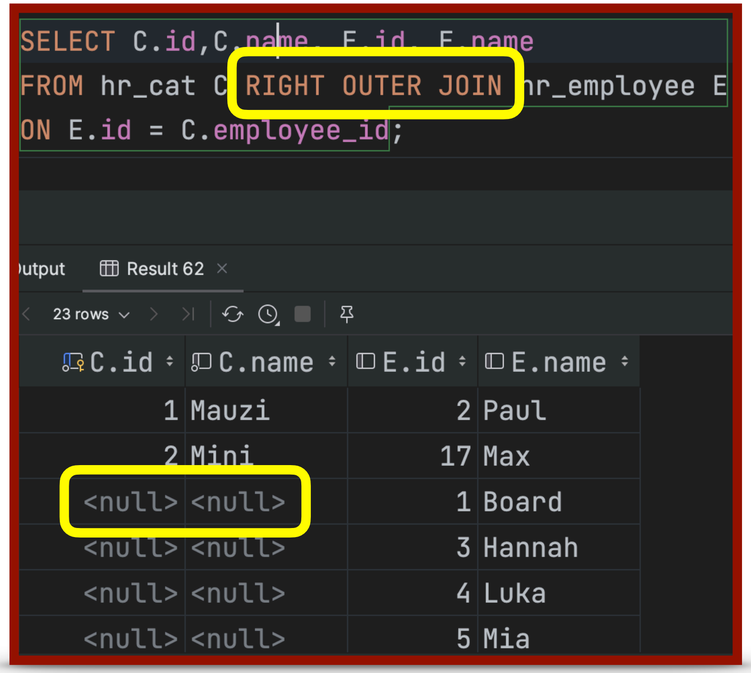
Outer Join

Example

- For the **left outer join**, **Roy** is included because all entities of hr_cat, left of the join, are considered, even if they are not in relation.
- The associated attributes of the non-existent partners are NULL.
- If we look at the same SQL command as a **right outer join**, all entities from hr_employee, right side of the join, appear and the associated attributes are NULL, if the employee is not related to a cat. Hence 'Roy' is not include.
- Please note that it is the side in the join that is important, not which of the attributes are output first.



db_unit_0x02_training



Outer Join

Example

- The **full outer join** is the union of the left outer join and the right outer join. This means that all entities on both sides are included, with NULL entries only if necessary.
- Some DBMS do not have a full outer join command, in which case the results of the two outer joins are simply combined with `union`.

Remarks

- Switching the tables in the join and the type of left or right outer join, results in the original outer join.
- Because inner joins only affect entities that have a partner, there is no left or right inner join.

```
SELECT C.id,C.name, E.id, E.name
FROM hr_cat C LEFT OUTER JOIN hr_employee E
ON E.id = C.employee_id
union
SELECT C.id,C.name, E.id, E.name
FROM hr_cat C RIGHT OUTER JOIN hr_employee E
ON E.id = C.employee_id;
```

utput | outer join (full)

24 rows

C.id	C.name	E.id	E.name
1	Mauzi	2	Paul
2	Mini	17	Max
3	Roy	<null>	<null>
<null>	<null>	1	Board
<null>	<null>	3	Hannah
<null>	<null>	4	Luka

db_unit_0x02_training

```
SELECT C.id,C.name, E.id, E.name
FROM hr_cat C LEFT OUTER JOIN hr_employee E
ON E.id = C.employee_id;

SELECT C.id,C.name, E.id, E.name
FROM hr_employee E RIGHT OUTER JOIN hr_cat C
ON E.id = C.employee_id;
```

Inner or Outer Join

Remarks

- Whether it is an inner or outer join can often be seen by looking at how it is worded.

Example

- **Only those** customers for whom an order exists, ... i.e. those who are in a relationship - thus inner join.
- **All customers for** whom an order has been placed **and those who have not yet** placed an order,... i.e. all customers, with or without partners, i.e. outer join.

```
SELECT O.id, O.customer_id, C.id, C.brand
FROM shop_order O INNER JOIN shop_customer C
ON O.customer_id= C.id;
```

utput | more examples

3 rows

O.id	customer_id	C.id	brand
3	7	7	E.ON
1	12	12	Sparkasse
2	12	12	Sparkasse

db_unit_0x02_training

```
SELECT O.id, O.customer_id, C.id, C.brand
FROM shop_order O RIGHT OUTER JOIN shop_customer C
ON O.customer_id=C.id;
```

utput | Result 73

13 rows

O.id	customer_id	C.id	brand
<null>	<null>	6	Börse
3	7	7	E.ON
<null>	<null>	8	Infinion

Natural Join

Intention

- For tables with attributes with the same name and the same meaning (!), the explicit specification of the condition (with `on`, `using` or `where`) for an inner join can be omitted. The DBMS then uses all attributes with the same name for a so-called **natural join**.

Remarks

- **As a rule of thumb, the use of natural joins is not recommended!**
- The DBMS always uses all attributes with the same name, i.e. adding new attributes or renaming existing ones can change the query!
- Note that extending a table is nothing unusual and can even happen automatically, for example when the object structure to be persisted is extended and the tables reflect the classes. This happens quickly, without taking into account possible natural joins - and then the queries do not return correct results (but technically still work).

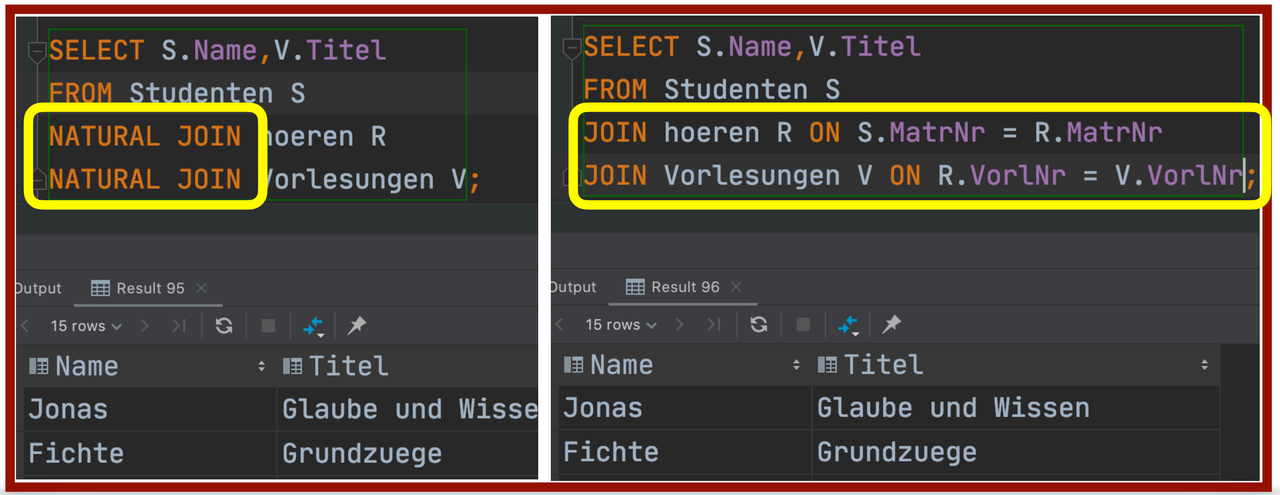
Natural Join

Remarks

- In the ami_zone database, natural joins are **not suitable** because, on the one hand, all primary keys are called id and all foreign keys <table>_id, i.e. these relationships must be specified explicitly, and on the other hand, all attributes with the same name, e.g. name, have different meanings, so they are also excluded.

Example

- In the example database ami_kemper, lectures are identified by a VorlNr and students by a MatrNr. The hoeren relationship relates MatrNr to VorlNr and a natural join can be used there.



db_unit_0x02_training

Check

Tasks

The required tables `shop_order` and `shop_customer` are taken from the `ami_zone` schema.

1. Query the **delivery time** and **customer name** for all orders:
   (a) Use an inner join with `where` once,
   (b) and an inner join with `(inner) join`.
   (c) Now add the name of the agent from `hr_employee`.

Delivery	Customer
13:14:00	Sparkasse
17:38:00	Sparkasse
08:22:00	E.ON

Result (a) and (b)

Delivery	Customer	Agent
13:14:00	Sparkasse	Anna
17:38:00	Sparkasse	Finn
08:22:00	E.ON	Anna

Result (c)

Check

Tasks

2. For each department (div_department), output the location (div_location). They are related to each other by div_located_in.

   (a) Use an (inner) join with join.

   (b) Also output departments that do not (yet) have a location.

   (c) Formulate (b) once as a left and once as a right outer join.

   (d) Use an SQL command to answer the question of which departments do not yet have a location.

	name	place
1	Marketing and Sales	Aachen
2	Research and Development	Aachen
3	Marketing and Sales	Jülich
4	Research and Development	Jülich
5	Marketing and Sales	Köln
6	Project Management	Köln
7	Board	Berlin
8	Human Resources	Berlin
9	Marketing and Sales	Berlin
10	Finance	Berlin
11	Project Management	Berlin

Result (a)

	name	place
1	Board	Berlin
2	Human Resources	Berlin
3	Marketing and Sales	Aachen
4	Marketing and Sales	Jülich
5	Marketing and Sales	Köln
6	Marketing and Sales	Berlin
7	Finance	Berlin
8	Research and Development	Aachen
9	Research and Development	Jülich
10	Project Management	Köln
11	Project Management	Berlin
12	Vehicle Fleet	<null>

Result (b) and (c)

	name	place
1	Vehicle Fleet	<null>

Result (d)

Check

Tasks

3. Now we are talking about employees (hr_employee), managers (boss) (foreign key employee_id in hr_employee), departments (div_department) and tasks (hr_task), related by hr_works_in_at.

   (a) Output all employees whose manager is Mia.

   (b) Print all employees with their name, department, task, hours per week and manager's name. Tip: Start with the relationship hr_works_in_at.

	name	Manager
1	Ben	Mia
2	Emma	Mia
3	Sofia	Mia
4	Jonas	Mia

Result (a)

	E1.name	D.name	T.name	hours...	Manager
15	Lena	Research and Develop…	Regular tasks	40.00	Leon
16	Leon	Project Management	Regular tasks	30.00	Ben
17	Leon	Project Management	Project Zerberus	10.00	Ben
18	Leonie	Research and Develop…	Regular tasks	20.00	Leon
19	Leonie	Research and Develop…	Education	20.00	Leon
20	Lilly	Marketing and Sales	Regular tasks	30.00	Sofia
21	Lilly	Research and Develop…	Project Zerberus	10.00	Sofia
22	Luis	Research and Develop…	Regular tasks	40.00	Lea
23	Luka	Board	Regular tasks	5.00	Board
24	Lukas	Research and Develop…	Regular tasks	40.00	Leon
25	Marie	Research and Develop…	Regular tasks	40.00	Lea
26	Max	Research and Develop…	Regular tasks	20.00	Lea

Partial result (b)

Check

Tasks

4. Analyse a customer network (shop_customer and shop_connected_to). Note that the knowledge 'customer a' knows 'customer b' is neither commutative nor transitive for this task.
   (a) Output the names of all connections of banks and savings banks (customer a has 'bank' or 'kasse' in its name) in the customer network.
   (b) Output all clients (client a) that have not yet established connections.

	C1.brand	C2.brand
1	Sparkasse	Börse
2	Deutsche Bank	Sparkasse
3	Commerzbank	Deutsche Bank
4	Commerzbank	Sparkasse

Result (a)

	brand
1	Börse
2	Infinion

Result (b)

Unit 0x03
Group Functions

Group Functions

Goal

- Grouping data according to specific criteria and aggregating data are important functions. These are commonly known as **Group Functions** or **Aggregation Functions**.

Group Functions

Intention

- Grouping data and using group functions.

Example

- Here, we calculate the listed aggregation functions for all products. These include Minimum, Maximum, Sum, Count, Average, and others.

```
SELECT P.name,P.price FROM shop_product P;
```

Output | # Use Group Functions

40 rows | Tx: Auto

	name	price
1	Spinach	1.99
2	Four Cheese Pizza	2.39
3	Spinach Pizza	2.29
4	Fish Fingers	1.99
5	Pasta Pan	3.29
6	Carrots	0.39

db_unit_0x03_training

```
SELECT min(price), max(price), sum(price), count(price),
       sum(price)/count(price), avg(price), stddev_pop(price),
       sqrt(var_pop(price)) FROM shop_product;
```

`min...	`max...	`sum...	`count...	`sum(price)/count...	`avg...	`stddev_pop...	`sqrt(var_pop
0.29	6.50	85.55	40	2.138750	2.138750	1.125476	1.12547

- stddev_pop refers to the so-called 'population standard deviation', in contrast to stddev_samp 'sample standard deviation'.

Group Functions

Intention

- Use Group Functions on selections.

```
SELECT * FROM shop_product WHERE category_id=1;
```

Output | Use Group Functions on selections.

5 rows | Tx: Auto | DDL

	id	name	category_id	unit	price	VAT
1	1	Spinach	1	PK	1.99	0.07
2	2	Four Cheese Pizza	1	PC	2.39	0.07
3	3	Spinach Pizza	1	PC	2.29	0.07
4	4	Fish Fingers	1	PK	1.99	0.07
5	5	Pasta Pan	1	PK	3.29	0.07

db_unit_0x03_training

Examples

- count(*) counts all lines including NULL (does not occur here)
- count(price) counts without NULL
- count(distinct price) counts only different prices

```
SELECT count(*), avg(price) FROM shop_product WHERE category_id=1;
```

	`count(*)`	`avg(price)`
1	5	2.390000

```
SELECT count(price), avg(price) FROM shop_product WHERE category_id=1;
```

	`count(price)`	`avg(price)`
1	5	2.390000

```
SELECT count(distinct price), avg(distinct price) FROM shop_product WHERE category_id=1;
```

	`count(distinct price)`	`avg(distinct price)`
1	4	2.490000

Remarks

- Consider whether to include NULL values or double values.

Group Functions

Intention

- Use Group Functions on grouped data.

Examples

- group by divides the total quantity into smaller groups, each used to determine the calculations.
- Refer to the previous example for count, min, max and avg.

```
SELECT count(price),category_id,min(price),max(price),avg(price) FROM shop_product
GROUP BY category_id;
```

Output | Use Group Functions on grouped data.

11 rows

	`count...	category_id	`min(price)`	`max(price)`	`avg(price)`
1	5	1	1.99	3.29	2.390000
2	4	2	0.29	1.29	0.737500
3	2	3	1.99	2.35	2.170000
4	3	4	0.79	0.99	0.920000
5	4	5	0.99	1.80	1.447500
6	3	6	2.79	3.55	3.276667
7	3	7	1.59	1.99	1.790000

db_unit_0x03_training

Group Functions

Intention

- Use Group Functions on grouped data of a selection.

Examples

- As before, but the total amount is first filtered to be in category 1, 2 or 4.

```
SELECT count(price),category_id,min(price),max(price),avg(price) FROM shop_product
WHERE category_id IN (1,2,4) GROUP BY category_id;
```

Output | Use Group Functions ... data of a selection.

3 rows

	`count...	category_id	`min(price)`	`max(price)`	`avg(price)`
1	5	1	1.99	3.29	2.390000
2	4	2	0.29	1.29	0.737500
3	3	4	0.79	0.99	0.920000

db_unit_0x03_training

having

Intention

- Use Group Functions on grouped data with condition.

Examples

- Condition **having** filters the subsets grouped with **group by**.
- This time, we are applying the condition to the **outcome of the grouping process**.
- Please note that there is often confusion over whether to use **where** or **having**.

```
SELECT count(price),category_id,min(price),max(price),avg(price) FROM shop_product
WHERE category_id IN (1,2,4) GROUP BY category_id HAVING min(price)>1;
```

Output | Use Group Functions ... data with condition.

1 row

	`count...	category_id	`min(price)`	`max(price)`	`avg(price)`
1	5	1	1.99	3.29	2.390000

db_unit_0x03_training

Grouped Attributes

Intention

- Use Group Functions correctly.

Remark

- If you group products by their category, the name and the unit of each product may differ. While it is technically feasible to include the name and unit in the select statement, it is nonsensical and may be disallowed in some DBMSs.
- Only use attributes that you know to be the same within the group. In some cases, you may need to limit the attributes to those specified in the 'group by' condition.

```
SELECT name, category_id, price, unit FROM shop_product;
```

Output | Use Group Functions correctly.

40 rows | Tx: Auto | DDL

	name	category_id	price	unit
1	Spinach	1	1.99	PK
2	Four Cheese Pizza	1	2.39	PC
3	Spinach Pizza	1	2.29	PC
4	Fish Fingers	1	1.99	PK

db_unit_0x03_training

```
SELECT name, category_id, price, unit FROM shop_product
GROUP BY category_id;
```

Output | ami_zone.shop_product

11 rows | Tx: Auto | DDL

	name	category_id	price	unit
1	Spinach	1	1.99	PK
2	Carrots	2	0.39	KG
3	Pork Sausage	3	1.99	PC
4	Milk	4	0.79	PC

having with Alias

Intention

- Use Group Functions with alias.

Examples

- An alias for `S` is given for the lowest value and applied in the `having` clause.
- The term `S` cannot be utilised in the `where` clause since the amount is still undetermined.
- Nevertheless, `S` is appropriate for sorting.

```
SELECT count(price),category_id,min(price) S FROM shop_product
WHERE category_id IN (1,2,4)
GROUP BY category_id HAVING 3*S>1
ORDER BY S;
```

Output | Use Group Functions with alias.

2 rows

	`count(price)`	category_id	S
1	3	4	0.79
2	5	1	1.99

db_unit_0x03_training

Multiple Attributes

Intention

- Use `group by` with multiple attributes.

Examples

- The table above shows how many items (-- n) per unit are subject to `VAT` (USt./MwSt.) to help compare.
- Then the products are grouped by `VAT` and then by `unit`.

```
SELECT count(*) FROM shop_product WHERE unit='KG' AND VAT=0.07;  --  4
SELECT count(*) FROM shop_product WHERE unit='PK' AND VAT=0.07;  -- 14
SELECT count(*) FROM shop_product WHERE unit='PC' AND VAT=0.07;  -- 16

SELECT count(id) FROM shop_product WHERE unit='PC' AND VAT=0.19; --  6
```

db_unit_0x03_training

```
SELECT count(VAT),VAT,count(unit),unit,min(price),max(price)
FROM shop_product GROUP BY VAT, unit ORDER BY VAT;
```

Output -- 6

4 rows

	`count(VAT)`	VAT	`count(unit)`	unit	`min(price)`	`max(price)`
1	4	0.07	4	KG	0.29	1.29
2	16	0.07	16	PC	0.79	6.50
3	14	0.07	14	PK	1.59	3.55
4	6	0.19	6	PC	0.99	3.20

Joins

Intention

- Use Group Functions with Joins.

Examples

- The top table displays a summary of all **drinks**. The second table groups them by product category.
- Technically, **C.name** should not be included in the **select** option, but this attribute is consistent within the **category**.

```
SELECT P.category_id, P.price, C.name FROM shop_product P
INNER JOIN shop_category C ON P.category_id= C.id
WHERE C.name LIKE '%drinks';
```

Output | Use Group Functions with Joins.

6 rows

	category_id	price	name
1	5	1.50	Cold Drinks
2	5	1.50	Cold Drinks
3	5	1.80	Cold Drinks
4	5	0.99	Cold Drinks
5	8	2.50	Hot Drinks
6	8	3.20	Hot Drinks

db_unit_0x03_training

```
SELECT count(P.price),P.category_id,avg(P.price),C.name FROM shop_product
INNER JOIN shop_category C ON P.category_id=C.id
WHERE C.name LIKE '%drinks' GROUP BY category_id;
```

Output | Result 45

2 rows

	`count(P.price)`	category_id	`avg(P.price)`	name
1	4	5	1.447500	Cold Drinks
2	2	8	2.850000	Hot Drinks

Group Functions

Overview

- `where` selects the total amount that can be grouped, while
- `group by` groups and applies aggregation functions as needed.
  Note that these cannot be used in the `where` condition.
- `having` is used to condition the results of grouping.
- Selected attributes should be included in the `group by` condition to ensure their uniqueness within the grouping.
- Aliases can be given to expressions to use in `having`.
- Joins can also be included in the grouping.

Check

Tasks

The required tables are taken from ami_zone.

1. Firstly, examine the employees' wage structure, who earn more than zero annually, and arrange them according to their manager's id.
   (a) Then, group the actual yearly pays according to the supervisor's ID, find the lowest and the highest pay in each group, count the number of salaries in each group and sort them. Verify the validity of the outcome.
   (b) Finally, include the average salary in the output and only produce entries where it exceeds 50,000.

	id	name	salary	employee_id	Boss
1	2	Paul	5000.00	1	Board
2	3	Hannah	5000.00	1	Board
3	4	Luka	5000.00	1	Board

	name	`count...	`min(M.salary)`	`max(M.sal...
1	Board	4	5000.00	110000.00
2	Mia	4	50000.00	90000.00
3	Ben	4	50000.00	70000.00

Result (a)

	name	`count...	`min...	`max...	avg
1	Mia	4	50000.00	90000.00	80000.000000
2	Ben	4	50000.00	70000.00	55000.000000
3	Emma	1	70000.00	70000.00	70000.000000

Result (b)

Check

Tasks

2. Questions regarding the product range.
   (a) What is the average price of a pizza and how many products are available?
   (b) How many products are in category 1 and 2? Also print the category name.
   (c) Calculate the number of products per category with a minimum unit price of over €2.
   (d) How many items are there priced between €1 and €2 per sales tax rate?

	pizza	avg
1	2	2.340000

Result (a)

	name	`count(P.id)`
1	Frozen Goods	5
2	Fruits, Vegetables	4

Result (b)

	name	`count(P.id)`	min
1	Barbecue Products	3	2.79
2	Hot Drinks	2	2.50

Result (c)

	`count(P.VAT)`	VAT
1	12	0.07
2	3	0.19

Result (d)

Check

Tasks

3. Firstly, examine the `hr_works_in_at` relationship in the DBMS.
   An employee may work in multiple departments
   in varying roles.
   Who are the employees working on more than
   one tasks within a single department?

   Advice: Start with simple queries and groupings,
   perhaps without joins, and gradually increase
   the complexity.
   Check the chosen examples to ensure plausibility.

	A.name	M.name	tasks
1	Finance	Emma	2
2	Research and Development	Leonie	2
3	Research and Development	Max	2
4	Research and Development	Finn	2
5	Project Management	Leon	2

Result (a)

Unit 0x04
Subselects

Subselects

Goal

- Use subqueries ➜ Subselects

Subselects

Intention

- Queries with conditions or operands that are unknown and must be determined by subqueries, the so-called subselects.

```
SELECT E.id, E.name FROM hr_employee E
WHERE E.name like 'Mia';
```

id	name
5	Mia

db_unit_0x04_training

Example

- Search for all employees whose manager is 'Mia'.
- Problem: The id of 'Mia' is not known, so it must first be determined. However, the absolute specification of a constant (here 5) is error-prone.
- Instead, we use a subselect. Its result is a value and the missing operand.

```
SELECT E.id, E.name FROM hr_employee E
WHERE E.employee_id = 5;
```

```
SELECT E.id, E.name FROM hr_employee E
WHERE E.employee_id = (
    SELECT id FROM hr_employee WHERE name='Mia'
);
```

id	name
6	Ben
7	Emma
8	Sofia
9	Jonas

Single-value subselects

Examples

- Find all employees who receive a higher annual salary than 'Jonas'.

```
SELECT E.id, E.name FROM hr_employee E
WHERE E.salary > (
    SELECT salary FROM hr_employee WHERE name='Jonas'
);
```

id	name
5	Mia
6	Ben
7	Emma
8	Sofia
13	Leon
14	Emilia

db_unit_0x04_training

- Find all employees of 'Mia' who receive a higher salary than 'Jonas'.

```
SELECT E.id,E.name FROM hr_employee E
WHERE E.salary > (
    SELECT salary FROM hr_employee WHERE name='Jonas'
) and E.employee_id = (
    SELECT id FROM hr_employee WHERE name='Mia'
);
```

id	name
6	Ben
7	Emma
8	Sofia

Single-value subselects

Intention

- Group functions (`min`) can also be used in subselects.

Example

- All employees who receive the minimum salary are found.
- Same task with `with` (not a subselect)

```sql
SELECT E.id,E.name,E.salary FROM hr_employee E
WHERE E.salary = (
  SELECT min(salary) FROM hr_employee
  WHERE salary>0
);

with Stats as (
    SELECT min(salary) as min FROM hr_employee
    WHERE salary>0
)
SELECT E.id,E.name,E.salary FROM hr_employee E, Stats
WHERE E.salary = Stats.min;
```

id	name	salary
2	Paul	5000.00
3	Hannah	5000.00
4	Luka	5000.00

db_unit_0x04_training

Subselects and IN

Intention

- Previously, subselects returned exactly one result, or in other words, one result row. Results consisting of multiple rows and multiple attributes can also be used with the IN, ANY and ALL operators.

Examples IN

- All employees will be found who receive a salary of 5000, 12000 or 18000.
- Find all employees who receive a minimum salary from the set of all minimum salaries grouped by manager.

```sql
-- IN
SELECT E.name, E.salary FROM hr_employee E
WHERE E.salary IN (5000,12000,18000);

-- group-functions with IN
SELECT E.name, E.salary FROM hr_employee E
WHERE E.salary IN (
    SELECT min(salary) FROM hr_employee
    GROUP BY employee_id
);
```

db_unit_0x04_training

Subselects and ANY

Examples ANY

- <ANY returns all products whose price is less than any price in product group 4, i.e. less than the maximum.
- >ANY returns all products whose price is greater than any price in product group 4, i.e. greater than the minimum.
- =ANY returns all products whose price is equal to any price of the products in product group 4, i.e. that is IN.

```sql
-- all (non-milk) products less than any milk product
SELECT P.name, P.price, P.category_id
FROM shop_product P WHERE P.price < ANY (
    SELECT price FROM shop_product WHERE category_id=4
) AND P.category_id <> 4;
-- all (non-milk) products less than max milk product
with Stats as (
    SELECT max(price) max FROM shop_product WHERE category_id=4
)
SELECT P.name, P.price, P.category_id
FROM shop_product P, Stats S WHERE P.price < S.max
    AND P.category_id <> 4;
```

db_unit_0x04_training

name	price	category_id
Carrots	0.39	2
Onions	0.29	2
Garlic	0.98	2
Bild	0.90	11

Subselects and ALL, NOT

Examples ALL, NOT

- <ALL returns all products whose price is less than all the prices of products in product group 4, i.e. less than the minimum.
- Similarly, >ALL returns all products whose price is greater than all the prices of the products in product group 4, i.e. greater than the maximum.
- NOT can be used with the IN, ANY and ALL operators.

```
SELECT P.name, P.price, P.category_id
FROM shop_product P WHERE P.price < ALL (
    SELECT price FROM shop_product WHERE category_id=4
) AND P.category_id <> 4;
```

db_unit_0x04_training

name	price	category_id
Carrots	0.39	2
Onions	0.29	2

Multi-value subselects

Intention

- Queries using multiple attributes in the subselect are also possible.

Example

- All products are found that have a price and a unit, such as at least one product from product group 1.
- Only items where both attributes match will be found.

```
SELECT P.name, P.unit, P.price, P.category_id FROM shop_product P
WHERE (unit, price) in (
    SELECT unit, price FROM shop_product
    WHERE category_id=1
) and category_id <> 1;
```

db_unit_0x04_training

name	unit	price	category_id
Chips	PK	1.99	7
Pasta in Sauce	PK	1.99	9

Inline Views

Intention

- In addition to using subselects in the WHERE or HAVING part, it is also possible to use so-called *inline views* for FROM or JOIN.

Example

- All product groups are displayed here, including an average price calculated from the products.
- The result set determined in the subselect is addressed under the alias Q and the average price is addressed there with A.
  Q behaves like a separate table or a separate so-called view (follows later).

```
SELECT C.*, Q.A FROM shop_category C, (
    SELECT P.category_id, avg(P.price) A
    FROM shop_product P GROUP BY category_id
) Q
WHERE C.id=Q.category_id;
```

db_unit_0x04_training

id	name	A
1	Frozen Goods	2.390000
2	Fruits, Vegetables	0.737500
3	Sausages, Cold Meat	2.170000
4	Milk Products	0.920000
5	Cold Drinks	1.447500
6	Barbecue Products	3.276667

Correlated Subselects

Intention

- Refer to the outer query in the subselects, the so-called *correlated subselects*.

Example

- This returns all products whose price is above the average of the corresponding category group.
- Note the reference to the outer query P.category_id in the subselect.

```
SELECT P.name, P.price, P.category_id
FROM shop_product P WHERE P.price > (
    SELECT avg(price) FROM shop_product
    WHERE category_id=P.category_id
);
```

db_unit_0x04_training

name	price	category_id
Pasta …	3.29	1
Bananas	1.29	2
Garlic	0.98	2
Meatba…	2.35	3
Quark	0.99	4
Yoghurt	0.88	4

Subselects and EXISTS

Intention

- Select data that depends on the existence of other data.

Example

- All employees who are also the manager of an employee are searched for.
- In principle, a SELECT 'X' FROM... in the subselect would be sufficient.
- NOT EXISTS is also possible.

```
SELECT E.id, E.name FROM hr_employee E
WHERE EXISTS (
    SELECT id FROM hr_employee
    WHERE hr_employee.employee_id=E.id
);
```

db_unit_0x04_training

id	name
1	Board
5	Mia
6	Ben
7	Emma
8	Sofia
12	Lea

Check

Tasks

The required tables are taken from ami_zone.

1. Which products have the same price as Chips?
   Chips should not appear in the output.
2. Which product groups have exactly the same number of products as product group 2?
   Print the category and the number of products.
3. Which products within a product group have the same price?
4. First look at the relationships div_department and hr_works_in_at.
   Which employee works on something like XCoin in finance?

Spinach	1.99
Fish Fingers	1.99
Pork Sausage	1.99
Pasta in Sauce	1.99

Result (1)

category_id	name	count
2	Fruits, Vegetables	4
5	Cold Drinks	4
9	Convenience Foods	4
10	Baked Goods	4

Result (2)

category_id	name	price
1	Spinach	1.99
1	Fish Fingers	1.99
5	Cola light	1.50
5	Nerd Bull	1.50

Result (3)

id	name	salary	employee_id	comment
5	Mia	110000.00	1	Board, CEO
7	Emma	90000.00	5	CHRO, CFO

Result (4)

Check

Tasks

5. Which employees work an average of 20 hours across all departments? Give their names and annual salaries.
6. First, look at the relationship shop_order. Which employees have already initiated an order (no subselect)?
7. Which products have not yet been ordered? If you have a correlated solution, try to think of an uncorrelated one as well.

employee_id	name	salary	avg
5	Mia	110000.00	20.000000
6	Ben	90000.00	20.000000
11	Finn	50000.00	20.000000
13	Leon	70000.00	20.000000
17	Max	12000.00	20.000000
20	Leonie	18000.00	20.000000

Result (5)

name	`for order`
Anna	1
Anna	3
Finn	2

Result (6)

id	name	category_id	unit	price	VAT
7	Onions	2	KG	0.29	0.07
8	Bananas	2	KG	1.29	0.07
9	Garlic	2	KG	0.98	0.07
10	Pork Sausage	3	PC	1.99	0.07
12	Milk	4	PC	0.79	0.07
13	Quark	4	PC	0.89	0.07

Result (7)

Check

Tasks

8. Use the shop_order, shop_consists_of, shop_product and shop_customer relations to create a total overview of all orders by summarising all products ordered per order.

`order`	customer_id	brand	sum
3	7	E.ON	42.4800
1	12	Sparkasse	50.7600
2	12	Sparkasse	83.8000

Result (8)

Unit 0x05
Schemes and Tables

Schemas and Tables

Goal

- Create, modify and delete schemas and tables in the DBMS.
- Preview: Create data in tables.

Schemas

Intention

- Query existing schemas or databases.

```
SHOW SCHEMAS;
SHOW DATABASES;

select * from information_schema.ENGINES;
```

Database
ami_algebra
ami_matse
ami_rel_model
ami_sport
ami_zone
information_schema
mysql
performance_schema

Remark

- In addition to our own schemas, there are always DBMS-specific information, such as information_schema, in which the DBMS keeps its own data, such as the different database engines.
- The management of schemas and internal structures is DBMS specific.

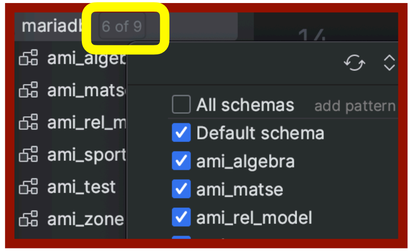
db_unit_0x05_training

ENGINE	SUPPORT	COMMENT	TRANSACTIONS
CSV	YES	Stores tables as CSV files	NO
MRG_MyISAM	YES	Collection of identical MyISAM tables	NO
MEMORY	YES	Hash based, stored in memory, useful…	NO
Aria	YES	Crash-safe tables with MyISAM herita…	NO
MyISAM	YES	Non-transactional engine with good p…	NO
SEQUENCE	YES	Generated tables filled with sequent…	YES
InnoDB	DEFAULT	Supports transactions, row-level loc…	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO

Managing Schemas

Intention

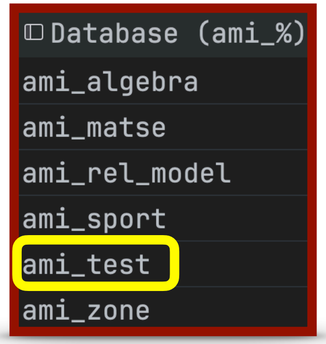
- Creating a schema, setting the default schema and deleting it.
- Caution! DBMS tools, such as DataGrip, may need to be refreshed to display changes!



context menu DataGrip

```
CREATE SCHEMA ami_test;
SHOW SCHEMAS LIKE 'ami_%';
USE ami_test;
DROP SCHEMA ami_test;
SHOW SCHEMAS;
```

db_unit_0x05_training



Remark

- **Caution! Deleting a schema will delete all tables and data!**
- Selecting a default schema is DBMS specific, e.g. PostgreSQL uses SET SEARCH_PATH = ami_test;

Tabels and Schemas

Intention

- Query existing tables in a schema.

Example

- The second command outputs only the tables that begin with 'shop'.
- This DBMS is case-insensitive.

```
SHOW TABLES FROM ami_zone;
SHOW TABLES FROM ami_zone like 'shop%';
SHOW TABLES FROM ami_zone like 'div%';
```

db_unit_0x05_training

Tables_in_ami_zone (shop%)
shop_category
shop_connected_to
shop_consists_of
shop_customer
shop_order
shop_product

Table Structure

Intention

- Query existing table structure, i.e. attributes/columns of a table.

Example

- Both commands produce the same result.
- The attributes are shown with properties, i.e. name, data type and other information.
  This will be explained again when creating.

```
SHOW COLUMNS FROM ami_zone.div_department;
DESCRIBE ami_zone.div_department;
```

db_unit_0x05_training

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	<null>	
name	varchar(100)	NO		<null>	
abbreviation	varchar(20)	NO		<null>	
head	varchar(100)	NO		<null>	

Create Table

Intention

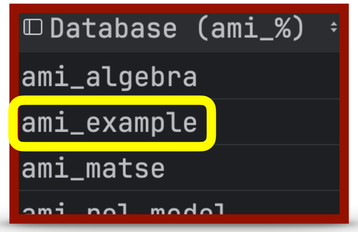
- Create tables.
- Caution: Before making structural changes to your own schemas and tables, it is better to make a backup.

Preparation

- In the following examples, work in a (temporary) schema, e.g. ami_example, which can be completely deleted afterwards.
- Do not make changes in ami_zone!
- Reminder: After programmatic changes to the table structures, refresh them if necessary.

```
CREATE SCHEMA ami_example;
SHOW SCHEMAS LIKE 'ami_%';
USE ami_example;
```

db_unit_0x05_training



Create Table

Example

- When creating the table, the individual columns or attributes are specified with their respective data types and properties such as `not null` or `default` values.
- For the data types `int`, `varchar`, etc., see the lecture notes or the DBMS documentation.
- `primary key` defines the key.
- `unique not null` must be unique and not be `NULL`.
- `default` sets default values, `now` generates a current timestamp (DBMS-specific).
- `boolean` is `tinyint(1)`, `true` is 1.

```
CREATE TABLE objects (
  id int primary key,
  name char(10) unique not null,
  comment varchar(255),
  number int(5),
  floating decimal(8,3) default 0.0,
  created datetime default now(),
  important boolean not null default true
);
SHOW COLUMNS FROM objects;
```

db_unit_0x05_training

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	<null>
name	char(10)	NO	UNI	<null>
comment	varchar(255)	YES		<null>
number	int(5)	YES		<null>
floating	decimal(8,3)	YES		0.000
created	datetime	YES		current_timestamp()
important	tinyint(1)	NO		1

Create Table

Example (Continuation)

- To see what the attribute properties do, records are created. The insert command follows, but the idea should be clear.
- The id must be specified explicitly, as well as attributes that are not allowed to be NULL.
- Attributes that are not assigned a value during insert either retain their default value or are set to NULL.

```
INSERT INTO objects (id,name) VALUES ('1','mueller');
INSERT INTO objects (id,name,number) VALUES ('2','meier','3');
SELECT * FROM objects;
```

db_unit_0x05_training

id	name	comment	number	floating	created	important
1	mueller	<null>	<null>	0.000	2023-12-07 19:32:11	1
2	meier	<null>	3	0.000	2023-12-07 19:32:12	1

Modify Tables

Intention

- Modify tables and columns or add, modify or delete attributes.

```
ALTER TABLE objects ADD (
  image blob,
  eps double default 0.01
);
SELECT * FROM objects;
SHOW COLUMNS FROM objects;
```

db_unit_0x05_training

Example

- The columns image and eps are added and eps is also given a default value.
- The data type blob can contain a large data object, e.g. a music file (binary large object).

id	name	com...	number	floating	created	imp...	image (UUID)	eps
1	mueller	<null>	<null>	0.000	2023-12-07 19:32:11	1	<null>	0.01
2	meier	<null>	3	0.000	2023-12-07 19:32:12	1	<null>	0.01

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	<null>
name	char(10)	NO	UNI	<null>
comment	varchar(255)	YES		<null>
number	int(5)	YES		<null>
floating	decimal(8,3)	YES		0.000
created	datetime	YES		current_timestamp()
important	tinyint(1)	NO		1
image	blob	YES		<null>
eps	double	YES		0.01

Modify Tables

Example

- The first command changes the data type and the default value.
- The second command also changes the name of the attribute.
- Since the eps and feps already contain values, the default value is changed, but not the value of the existing entities.

```
ALTER TABLE objects MODIFY eps float default 0.002;
ALTER TABLE objects CHANGE eps feps float default 0.003;
SHOW COLUMNS FROM objects;
```

db_unit_0x05_training

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	<null>
name	char(10)	NO	UNI	<null>
comment	varchar(255)	YES		<null>
number	int(5)	YES		<null>
floating	decimal(8,3)	YES		0.000
created	datetime	YES		current_timestamp()
important	tinyint(1)	NO		1
image	blob	YES		<null>
feps	float	YES		0.003

Modify Tables

Example

- The feps and image attributes are deleted. The other attributes are not affected.
- In some DBMS, attributes can be marked as unused.

```
ALTER TABLE objects DROP feps;
ALTER TABLE objects DROP image;
SHOW COLUMNS FROM objects;
```

db_unit_0x05_training

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	<null>
name	char(10)	NO	UNI	<null>
comment	varchar(255)	YES		<null>
number	int(5)	YES		<null>
floating	decimal(8,3)	YES		0.000
created	datetime	YES		current_timestamp()
important	tinyint(1)	NO		1

Delete Tables

Intention

- Rename, clear and delete tables.

Example

- The first command renames the table objects to elements.
- The TRUNCATE command deletes all records, but leaves the table itself intact.
- Attention! Deleting a table with DROP also deletes all the data in the table!

```sql
-- Rename table
ALTER TABLE objects RENAME TO elements;
SELECT * FROM elements;

-- Remove all elements
TRUNCATE TABLE elements;
SELECT * FROM elements;

-- Remove table itself
DROP TABLE elements;
SHOW TABLES FROM ami_example;
```

db_unit_0x05_training

Table Constraints

Intention

- Tables with table constraints and foreign keys, that is, with attributes that refer to records in other tables.

Example

- The example models people and their pets, where a person can be responsible for several pets.
- The person table is referenced by the pet table below.
- When creating, the PRIMARY KEY is not added to the attribute, but at the end.

```sql
-- Create tables with table constraints
CREATE TABLE person (
person_id INT NOT NULL,
name VARCHAR(50) NOT NULL,
PRIMARY KEY (person_id)
);

-- Some data
INSERT INTO person (person_id,name) VALUES ('11','MIA');
INSERT INTO person (person_id,name) VALUES ('12','LEA');
SELECT * FROM person;
```

db_unit_0x05_training

person_id	name
11	MIA
12	LEA

Table Constraints

Example (Continuation)

```
-- Create table with foreign key and index
CREATE TABLE pet (
pet_id INT NOT NULL,
name VARCHAR(50) NOT NULL,
person_id INT NULL,
PRIMARY KEY (pet_id),
INDEX person_idx (person_id ASC),
CONSTRAINT fk_person FOREIGN KEY
(person_id) REFERENCES person (person_id) );

-- Some data and a join
INSERT INTO pet (pet_id,name,person_id) VALUES ('1','Wuff','11');
INSERT INTO pet (pet_id,name) VALUES ('2','Bello');
SELECT * FROM pet M LEFT OUTER JOIN person F ON M.person_id=F.person_id;
```

- person_id is the foreign key pointing to the attribute of the same name in the person table. The value can also be NULL if there is no person (yet) (as in Bello).
- The constraint is called fk_person.
- With some DBMS there has to be an index for each foreign key relationship, e.g. to speed up joins, here person_idx.

db_unit_0x05_training

pet_id	M.name	M.person_id	F.person_id	F.name
1	Wuff	11	11	MIA
2	Bello	<null>	<null>	<null>

- When creating data in pet, an optional foreign key can be specified. This can be NULL or refer to a person.
- The DBMS, more precisely the constraint, prevents the specification of a non-existing record.

Table Constraints

Example (Continuation)

- A check constraint has been added to ensure that a pet name is not longer than 10 characters. Then an insert would fail, e.g. this one.
- It is also possible to specify this check as a table constraint.

```
CREATE TABLE pet (
pet_id INT NOT NULL,
name VARCHAR(10) NOT NULL CHECK (LENGTH(name) <= 10),
person_id INT NULL,
PRIMARY KEY (pet_id),
INDEX person_idx (person_id ASC),
CONSTRAINT fk_person FOREIGN KEY
(person_id) REFERENCES person (person_id) );
```

```
INSERT INTO pet (pet_id,name,person_id) VALUES ('3','Mr. Robinson','11');
```

```
CREATE TABLE pet2 (
pet_id INT NOT NULL,
name VARCHAR(10) NOT NULL,
person_id INT NULL,
PRIMARY KEY (pet_id),
INDEX person_idx (person_id ASC),
CONSTRAINT nameLengthCheck CHECK (LENGTH(name) <= 10),
CONSTRAINT fk_person2 FOREIGN KEY
(person_id) REFERENCES person (person_id) );
```

db_unit_0x05_training

Remark

- There are more constraints, e.g. a referential integrity constraints such as on delete cascade. This constraint is used, for example, to delete existence-dependent details in a 1:n relationship (without example).

Check

Tasks

The required tables are *not* taken from ami_zone.

1. Create your own schema ami_sport and select it as default schema.
2. In ami_sport there are athletes, competitions and teams with attributes. In addition, there is a relationship attends describing who is in which team for which competition, and a relationship referees describing who is the referee for that (for cost reasons there are no own referees, but there is always exactly one athlete who referees an event).

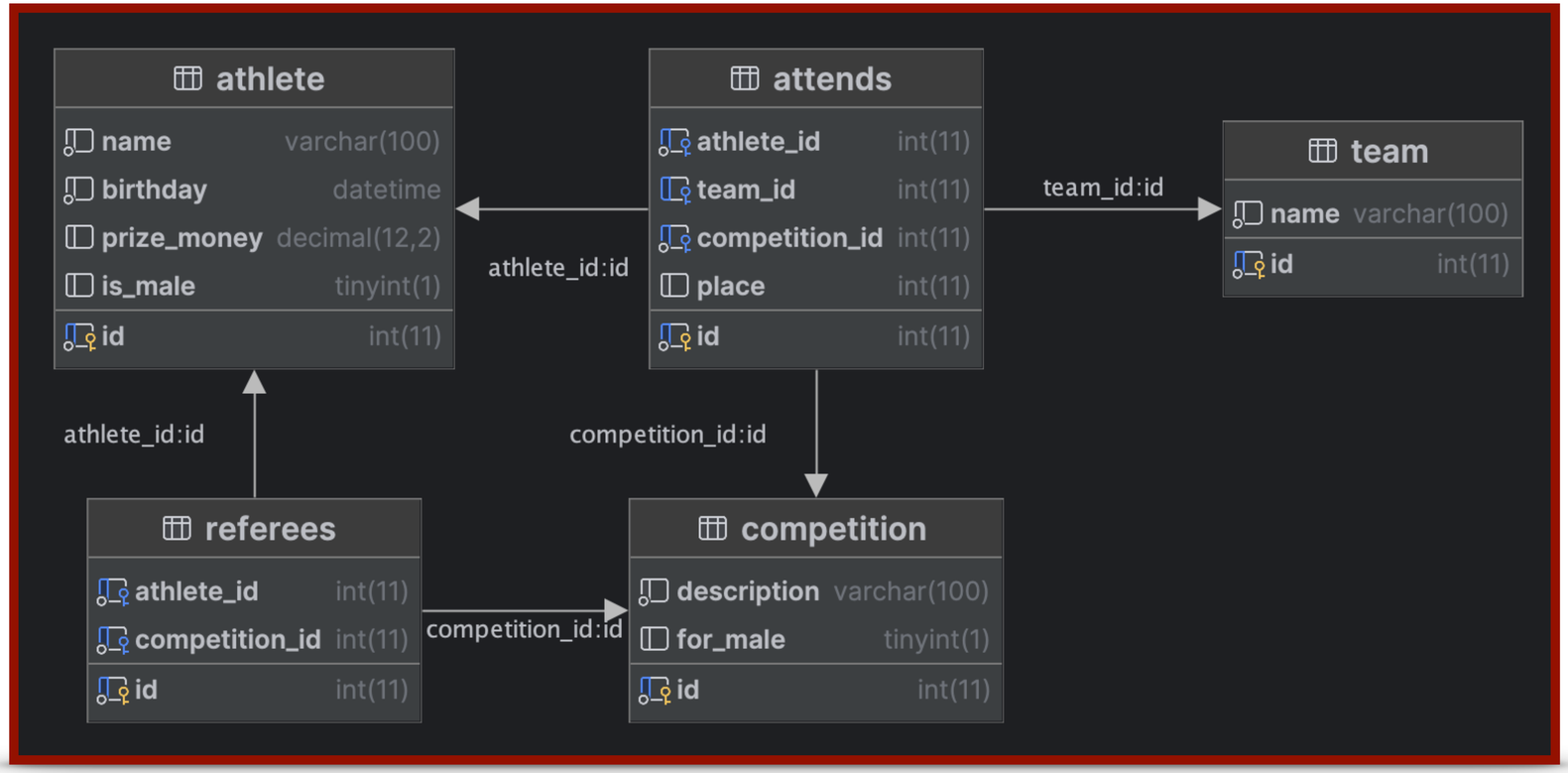
   Use SQL commands to create the tables shown in the following diagram with the correct foreign key relationships. Start with the simple tables athlete, competition and team and then create the relationships referees and attends with the foreign key relationships.

   The explanation of the content follows.

Check

Tasks

2. The ER diagram is shown on the right. Here is worth mentioning:
   - A small golden key is a primary key.
   - A small blue key is a foreign keys.
   - A small circle indicates *non null*.
   - Attributes like is_male or for_male are coded as follows: male (true), female (false), divers or mixed (null).
   - Default values are not shown, select as appropriate.



Check

Tasks

2. The following conditions apply:
   - By default, the prize money is set to 0 and the athlete is male.
   - A competition has a non-optional name and is for men, women or mixed; by default it is for men.
   - A team has only one name.
   - n Athletes can participate in m competitions. If it is a team event, the team is also specified.
   - As before, an INSERT SQL command always has a form (e.g. for two columns):
     `INSERT INTO 'table' (attribute1, attribute2) VALUES ('value1', 'value2');`

     It is also possible to insert several data with one command:
     `INSERT INTO 'table' VALUES ('attribute1','attribute2'),('attribute1', 'attribute2');`

Check

Tasks

2. Insert SQL commands can be taken from the SQL script db_unit_0x05_training. **You do not need to type them.**

   However, your table structures must be suitable.

id	name	birthday	prize_money	is_male
101	Anna	1990-02-01 00:00:00	0.00	0
102	Olga	1991-03-01 00:00:00	0.00	0
111	Enie	1992-04-01 00:00:00	100.00	0
112	Antje	1993-05-01 00:00:00	200.00	0
121	Boris	1990-06-01 00:00:00	3000.00	1
122	Ivan	1991-07-01 00:00:00	4000.00	1

athlete

id	description	for_male
56	Tennis Preliminary Round - Doubles	0
98	Tennis Final - Singles	1
99	Tennis Final - Singles	0

competition

id	athlete_id	team_id	competition_id	place
1	101	98765	56	<null>
2	102	98765	56	<null>
3	111	12345	56	<null>
4	112	12345	56	<null>
5	101	<null>	99	<null>
6	111	<null>	99	<null>
7	121	<null>	98	<null>
8	122	<null>	98	<null>

attends

id	athlete_id	competition_id
1	121	56
2	122	99
3	101	98

referees

id	name
12345	Team NL
98765	Team PL

team

Check

Tasks

Answer the following questions:

3. Which athlete participates in more than one event?
   Give the name and number of competitions for each athlete.
4. What athletes do not referee competitions?
   Give the names.
5. Which teams take part in which events?
   Give the names of the teams and the competitions.
6. Which athletes are in a final?
   Give the names and the names of the finals.

id	name	`count(*)`
101	Anna	2
111	Enie	2

Result (3)

id	name
102	Olga
111	Enie
112	Antje

Result (4)

name	description
Team NL	Tennis Preliminary Round - Doubles
Team PL	Tennis Preliminary Round - Doubles

Result (5)

id	...	description	for_male
121	Boris	Tennis Final - Singles	1
122	Ivan	Tennis Final - Singles	1
101	Anna	Tennis Final - Singles	0
111	Enie	Tennis Final - Singles	0

Result (6)

Unit 0x06
Preparations: Time and Date, Views, Transactions

Time and Date functions

Goal

- Use time and date functions.

Remark

- The way time and date are handled is a constant source of delight, because there are so many variants, data types and DBMS-specific settings.

  However, we want to give at least a few (MySQL/MariaDB specific) examples here, so that you can get an idea of how these functions can be used. For your own DBMS, always look it up.

Name	Description
ADDDATE()	Add time values (intervals) to a date value
ADDTIME()	Add time
CONVERT_TZ()	Convert from one time zone to another
CURDATE()	Return the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Return the current time
DATE()	Extract the date part of a date or datetime expression
DATE_ADD()	Add time values (intervals) to a date value
DATE_FORMAT()	Format date as specified
DATE_SUB()	Subtract a time value (interval) from a date
DATEDIFF()	Subtract two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Return the name of the weekday
DAYOFMONTH()	Return the day of the month (0-31)
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
EXTRACT()	Extract part of a date
FROM_DAYS()	Convert a day number to a date
FROM_UNIXTIME()	Format Unix timestamp as a date
GET_FORMAT()	Return a date format string
HOUR()	Extract the hour

Extracted date functions from MySQL

Time Zones

Background

- Timestamps are generally dependent on time zones, which are either explicitly specified or implicitly used by the system or DBMS. You can see what these look like at https://de.wikipedia.org/wiki/Zeitzone .
- The local times are relative to the Coordinated Universal Time (UTC), where UTC±0 is the time zone time related to the Greenwich meridian. The differences between Universal Time (UT) and UTC are not discussed here.
- Furthermore, summer and winter time have to be considered. In German-speaking countries, MEZ (UTC+1h) is the standard time in winter, while Central European Summer Time (MESZ, UTC+2h) is used in the summer months.

Extracted date functions from MySQL

Internal representation

Background

- How a point in time is represented internally is a topic in its own right. Generally, one could think that this initially plays no role. However, if you need to store very precise, older or distant times, you need to make sure that the internal format can handle it.

  An example of this is the so-called Unix time. This counts the past seconds since Thursday, 1. January 1970, 00:00 UTC (the epoch).
  On 19. January 2038 at 3:14:08 UTC, systems which store the Unix time in a 32-bit variable with a leading sign will experience a jump…

- YEAR: A one-byte integer
- DATE: A three-byte integer packed as YYYY×16×32 + MM×32 + DD
- TIME: A three-byte integer packed as DD×24×3600 + HH×3600 + MM×60 + SS
- TIMESTAMP: A four-byte integer representing seconds UTC since the epoch ('1970-01-01 00:00:00' UTC)
- DATETIME: Eight bytes: A four-byte integer for date packed as YYYY×10000 + MM×100 + DD and a four-byte integer for time packed as HH×10000 + MM×100 + SS

Extracted from MySQL documentation storage format.

Internal representation

Background

- All times provided by any system have to be checked with regard to the own application, e.g:
  - Do I know what the system/function provides?
    (UTC vs. local time including time shift)
  - Are local times sufficient for my problem?
    (current cinema programme vs. internationally coordinated rocket launch)
  - What is the time period?
    (today +/- a few years vs. Christmas in the Dark Ages)
  - What accuracy is needed? Do I need only days, or also seconds, or even more precise time stamps?
    (cinema programme vs. measurement data)

Time Zones

Example

- Preface: These settings are MySQL/MariaDB specific and what we show here is not complete. It also depends on the configuration of the system, specifically whether MySQL/MariaDB is configured so that timezones are available as names.
- There is a local (session) and a global (global) timezone, which can be set to either a value like 'Europe/Berlin' or an offset, e.g. '+02:00'.
  We will now only look at the local time.
  You can see the effect in the example, where @@session.time_zone is the local timezone and **now()** returns the current date and time.

```sql
-- UTC
SET SESSION time_zone = 'SYSTEM';
SELECT @@session.time_zone, now();

-- Germany
SET SESSION time_zone = 'Europe/Berlin';
SELECT @@session.time_zone, now();

-- with offset
SET SESSION time_zone = '+02:00';
SELECT @@session.time_zone, now();
```

db_unit_0x06_training

@@session.time_zone	now()
SYSTEM	2023-12-15 19:50:46
@@session.time_zone	now()
Europe/Berlin	2023-12-15 20:51:18
@@session.time_zone	now()
+02:00	2023-12-15 21:51:33

Every now and then

Examples

- Get current date and time (datetime type) and convert to date and time type.

```
SELECT now() 'N',
       cast(now() AS date) 'D',
       cast(now() AS time) 'T',
       cast(now() AS datetime) 'DT';
```

N	D	T	DT
2023-12-15 20:57:03	2023-12-15	20:57:03	2023-12-15 20:57:03

- A date can be modified, either by using functions like date_add or by using + or -.

```
SELECT cast(now() AS date) + interval 2 year;
SELECT cast(date_add(now(), interval 2 year) as date);
SELECT cast(now() AS time) - interval 2 hour;
```

db_unit_0x06_training

- A period of time can also be determined, here in days or hours.

```
SELECT datediff(date_add(now(), interval 1 year),now());
SELECT timediff(NOW(), UTC_TIMESTAMP);
```

- Hint: Or a day of week: dayofweek...

Conversions

Examples

```
SELECT str_to_date('13.10.2014',get_format(date,'EUR'));
SELECT date_format('2014-10-13',get_format(date,'EUR'));
SELECT date_format('2014-10-13','%W %M %Y');
```

- The first expression converts a string given in European format to a date.

`str_to_date('13.10.2014',get_format(date,'EUR'))`
2014-10-13

- The second expression converts a date to the European format.

`date_format('2014-10-13',get_format(date,'EUR'))`
13.10.2014

db_unit_0x06_training

- The format string can also be given directly, here in the last example with the day of the week, month and 4-digit year.

`date_format('2014-10-13','%W %M %Y')`
Monday October 2014

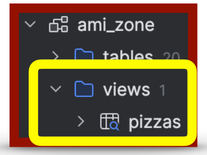
Views

Goal

- Create and delete views.

Example

- The query filters all the pizzas in the range. We want to formulate this as a view with the name pizzas.
- If pizzas already exists, it is redefined and then used like a table.
- pizzas can then also be found as a view in ami_zone.
- To delete pizzas, use drop view.




```
SELECT P.name, P.price AS 'price'
    FROM shop_product P where P.name like '%pizza%';
```

name	price
Four Cheese Pizza	2.39
Spinach Pizza	2.29

db_unit_0x06_training

```
CREATE OR REPLACE VIEW pizzas AS (
    SELECT P.name, P.price AS 'price'
    FROM ami_zone.shop_product P where P.name like '%pizza%'
);
```

```
SELECT * FROM pizzas;
```

name	price
Four Cheese Pizza	2.39
Spinach Pizza	2.29

```
DROP VIEW pizzas;
```

Transactions

Goal

- Using Transactions.
- Use database operations only as a whole or discard them, cf. Account movement at a bank.
- See ACID or atomicity.

Transactions

Preparations

- To understand the effects of the commit and rollback commands associated with a transaction, we need a simple starting situation that we can easily recreate.
- To do this, we use the ami_example schema and the person and pet tables from db_training_0x05. In the script, the generating commands are also specified. The initial data looks like this:

person_id	name
11	MIA
12	LEA

pet_id	name	person_id
1	Wuff	11
2	Bello	<null>

pet_id	M.name	M.person_id	F.person_id	F.name
1	Wuff	11	11	MIA
2	Bello	<null>	<null>	<null>

db_unit_0x06_training

So from here we assume that tables and data are available.

Transactions

Preparations

- There is also a MySQL/MariaDB setting that causes every command to be processed immediately. This setting is normally active. However, to be on the safe side, we set it to 1 in the script.

```
SET autocommit=1;
```

db_unit_0x06_training

- Especially in DataGrip it is the case that loaded scripts are executed in a certain session. This is assigned manually or automatically when the script is loaded. We also need another session (console), which can be opened from the context menu.
- We now add and change data in the tables. We look at the different sessions to see what is there, i.e. what can be seen from 'outside'.
- We can then commit the changes or rollback to the original state.

Transactions

Example 1

```
START TRANSACTION;
INSERT INTO person (person_id,name) VALUES (21,'Max');
SELECT * FROM person;
```

- We start with our initial data and a transaction. Then we insert Max and check the result.
- You can see that the change is immediately visible in the first session in which the script is executed, but not in the other.
  This is also the view of another user.
- Now, a rollback causes all the changes to be undone.

person_id	name
11	MIA
12	LEA
21	Max

db_unit_0x06_training

person_id	name
11	MIA
12	LEA

other console

Example 2

```
ROLLBACK;
SELECT * FROM person;
```

person_id	name
11	MIA
12	LEA

db_unit_0x06_training

- Do the same as before, but now use commit instead of rollback. This causes all changes to be visible in all sessions, and thus for all users.
  Attention, delete the new record before the next example.

```
COMMIT;
SELECT * FROM person;
```

person_id	name
11	MIA
12	LEA
21	Max

db_unit_0x06_training

person_id	name
11	MIA
12	LEA
21	Max

other console

Transactions

Example 3

- We start again with our initial data and a transaction.
- Now we use a wrong person_id=22, which leads directly to an error because the foreign key is missing.

```
START TRANSACTION;
INSERT INTO person (person_id,name) VALUES (21,'Max');
-- attention: person_id 22 is not existing
INSERT INTO pet (pet_id,name,person_id) VALUES (5,'Mini',22);
```

db_unit_0x06_training

```
[23000][1452] (conn=4) Cannot add or update a child row: a foreign key constraint fails (`ami_example`.`pet`, CON
```

- Using a transaction and restoring via rollback in case of an error does not fix the error, but for everyone else the database is still in its original state. You also don't have half-filled-in data, but all or nothing – which was exactly the goal.

```
ROLLBACK;
```

Transactions

Example 4

```
START TRANSACTION;
-- attention: the person_id 21 is still to come, therefore error
INSERT INTO pet (pet_id,name,person_id) VALUES (5,'Mini',21);
INSERT INTO person (person_id,name) VALUES (21,'Max');
ROLLBACK;
```

db_unit_0x06_training

- We start again with our initial data and a transaction.
- In the example 3 above, a non-existing foreign key was deliberately used, but this error situation also occurs when tables from external sources are imported into the DBMS in the 'wrong' order. To solve the problem, one can try to find a logical order, so that the data is built up gradually and all references are valid when imported. However, if there are cross-references between the tables, this is not possible. Then you can switch off the check for referential integrity for a 'short' moment, insert the data and get around this problem. Example 4 shows exactly this.

```
SET foreign_key_checks = 0;
```

- If you forget to reset the foreign_key_checks or are not careful, you can insert unknown foreign keys without noticing it at first...
  So don't forget to restore the data.

```
SET foreign_key_checks = 1;
```

Transactions

Example 5

- In short, the autocommit=1 setting mentioned in the preamble means that every command issued is immediately committed.
  This is why a data change is immediately visible in another session.

```
SET autocommit=0;
-- without autocommit, the transaction is still open.
INSERT INTO person (person_id,name) VALUES (21,'Max');
-- check table in another session
SELECT * FROM person;
ROLLBACK;
SELECT * FROM person;
SET autocommit=1;
```

db_unit_0x06_training

- If the setting is turned off, a commit must be made before the changes are visible in other sessions.
  The system behaves as if a transaction is constantly active, which must first be confirmed or rolled back.
- Caution: This effect is (on my system) only visible with MySQL. In MariaDB the setting can be set, but the system still behaves as if autocommit=1.

Check

Tasks (Date and Time)

Use the/your competition scheme ami_sport from the previous tasks. Then answer the following questions.
(See task 5.2 in db_solution_0x05.sql if you need the schema for SQL commands to create the tables and populate the data.)

1. Which athletes were born on a Monday? Give their name and date of birth. Check the help page of your DBMS for appropriate functions.

name	birthday
Ivan	1991-07-01

2. Which athletes are younger than 35? Give their names and ages.
   Note: You do not need to consider leap years.

   (a) Rewrite your solution so that the age calculation only appears once in the SQL statement.

name	age
Anna	33
Olga	32
Enie	31
Antje	30
Boris	33
Ivan	32

Results

3. Which competition has the highest average age of its competitors? Specify the name of this race with the average age.

description	max_avg
Tennis Final - Singles	33.02600000

Check

Tasks (Views)

The required tables are taken from ami_zone.

4. We look at products and get closer to a working select command, which we then wrap up in a view.

   (a) Warm-up: Search for all products in category 1 and output the description and the unit price.

   (b) Extend (a) and search not directly for products in category 1, but for products in category Frozen Goods. Use a subselect to do this.

   (c) Rewrite (b) so that the product group Frozen Goods is found using with.

   (d) Create a view only_fg with one of the three queries (a)-(c). Use it in a select and delete it at the end.

Results

name	price
Spinach	1.99
Four Cheese Pizza	2.39
Spinach Pizza	2.29
Fish Fingers	1.99
Pasta Pan	3.29

Check

Tasks

The required tables are taken from ami_zone.

5. We look at orders and approach a select, which we then wrap up in a view.
   (a) Take a look at the shop_order and shop_consists_of tables and understand how an order is made up of a number of individual products.
   (b) Together with the product price, you can calculate the total price of each order. Formulate an appropriate select.

order_id	summe
1	50.7600
2	83.8000
3	42.4800

Results

   (c) Add the customer name and order date to your selection from (b), create a view from it, use it and delete it.

id	vom	brand	sum
1	2023-12-17	Sparkasse	50.7600
2	2023-12-17	Sparkasse	83.8000
3	2023-12-17	E.ON	42.4800

   (d) Depending on the DBMS or version, problems may occur if the view contains an embedded view, e.g. a subselect. For this case, and/or if your view from (c) contains such a subselect, re-formulate your view so that several nested views are used.

Check

Tasks (Transaction)

You are free to choose the tables you need.

6. Create or use a schema and a table of your choice. Start a transaction. Now add data using `insert` (similar to the one in `ami_sport`). Then either discard the changes (`rollback`) or accept them (`commit`). Check the result in another session/console.

Unit 0x07
Create, change and delete data

C(R)UD Functions

Goal

- Create or insert, modify or update and delete data.

Preparation

- Use the/your competition scheme `ami_sport` from the previous tasks.
  See task `5.2` in `db_solution_0x05.sql` if you need the schema for SQL commands to create the tables and populate the data.

Insert Data

Intention

- insert new data.

```
INSERT INTO athlete (id, name, birthday, is_male)
VALUES ('131', 'Pierre', '1991-12-24', '1');
INSERT INTO athlete (id, name, birthday, is_male)
VALUES (132, 'Rob', '1991-12-24', true);
INSERT INTO athlete (id, name, birthday)
VALUES (133, 'Pete', cast(now() AS date) - interval 20 year);
```

db_unit_0x07_training

id	name	birthday	prize_money	is_male
131	Pierre	1991-12-24 …	0.00	1
132	Rob	1991-12-24 …	0.00	1
133	Pete	2004-01-04 …	0.00	1

Example

- Here three records are created one after the other. Arguments can either be enclosed in quotation marks or, if the type allows, without them.
- The values can also be the result of an expression, e.g. the date of birth of Pete.

Remark

- In order to recover the original data quickly, all operations are performed in one transaction, which we rollback.

Insert Data

Intention

- Generate attribute values from other data.

Example

- Here, the birthdate for Marc is derived from the birthdate of Pete using a subselect (twins at midnight).
- The result of the subselect must be one value.

```
INSERT INTO athlete (id, name, birthday)
VALUES (134, 'Marc', (
  SELECT birthday: S.birthday+interval 1 day FROM athlete S
  WHERE S.name='Pete'
));
```

db_unit_0x07_training

id	name	birthday	prize_money	is_male
133	Pete	2004-01-04	0.00	1
134	Marc	2004-01-05	0.00	1

Insert Data

Intention

- Create multiple records at once.

Example

- Specifying multiple value tupels will also create multiple records.
- You can also specify `NULL` if this is allowed for the particular attribute.

```
INSERT INTO athlete (id, name, birthday, prize_money) VALUES
  (135, 'Tick', '1992-02-21 15:00', 100.0),
  (136, 'Trick', '1992-02-21 15:01', NULL),
  (137, 'Track', '1992-02-21 15:02', 200.0);
```

db_unit_0x07_training

id	name	birthday	prize_money	is_male
134	Marc	2004-01-05 …	0.00	1
135	Tick	1992-02-21 …	100.00	1
136	Trick	1992-02-21 …	<null>	1
137	Track	1992-02-21 …	200.00	1

Insert Data

Intention

- Generate entire records from other records.

Example

- It is also possible to generate new records from a subselect when the result consists of multiple rows and attributes, similar to single attribute generation.
- In this case VALUES is omitted.
- This makes it easy to use tables as 'templates' for derived data.

```
INSERT INTO athlete (id, name, birthday)
    SELECT id: S.id+3, name: concat(S.name,' Double'),S.birthday
    FROM athlete S WHERE S.name like '%ck';
```

db_unit_0x07_training

id	name	birthday	prize_money	is_male
135	Tick	1992-02-21 …	100.00	1
136	Trick	1992-02-21 …	<null>	1
137	Track	1992-02-21 …	200.00	1
138	Tick Double	1992-02-21 …	0.00	1
139	Trick Double	1992-02-21 …	0.00	1
140	Track Double	1992-02-21 …	0.00	1

Update Data

Intention

- Modify or update records.

Example

- Here the prize money of the record with id 138 is changed to 1000.
  The following select shows that this is the attribute that has been changed.

```
UPDATE athlete SET prize_money=1000 WHERE id=138;
SELECT S.id, S.name ,S.prize_money, S.is_male
FROM athlete S WHERE S.id>=138;
```

db_unit_0x07_training

id	name	prize_money	is_male
138	Tick Double	1000.00	1
139	Trick Double	0.00	1
140	Track Double	0.00	1

Update Data

Intention

- Change multiple attributes at once.

Example

- It is also possible to modify several records and several attributes with one command.
- This will increase the prize money by 50 and set the status is_male to false (=0).
- Caution: If where is omitted, *all* records will be changed. Some tools, e.g. DataGrip, can prevent this automatism by settings, e.g. 'Unsafe Query' or 'Safe Updates' or similar.

```
UPDATE athlete
SET prize_money=prize_money+50, is_male=false
WHERE id>=138;
```

db_unit_0x07_training

id	name	prize_money	is_male
138	Tick Double	1050.00	0
139	Trick Double	50.00	0
140	Track Double	50.00	0

Update Data

Intention

- Modify records using **subselects**.

Example

- This sets the **prize money** to a value determined by the **subselect**.
- If the result is **NULL**, the attribute will also be set to **NULL**.
- Caution: If the **subselect** returns more than one record, there is an error.

```
UPDATE athlete
SET prize_money = (
    SELECT max(prize_money % 1234) as 'max_prize_money'
    FROM athlete
) WHERE id=139;
SELECT * FROM athlete WHERE id=139;
```

db_unit_0x07_training

```
SELECT id, name, prize_money, prize_money % 1234 FROM athlete;
```

id	name	prize_money	`prize_money % 1234`
137	Track	200.00	200.00
138	Tick Double	1050.00	1050.00
139	Trick Double	1050.00	1050.00
140	Track Double	50.00	50.00

- Caution: When modifying records using **subselects** that refer to the table being modified, there are problems with some **MySQL** versions.

Delete Data

Intention

- Delete records.

```
DELETE FROM athlete WHERE id>130;
```

db_unit_0x07_training

id	name	birthday	prize_money	is_male
111	Enie	1992-04-01 …	100.00	0
112	Antje	1993-05-01 …	200.00	0
121	Boris	1990-06-01 …	3000.00	1
122	Ivan	1991-07-01 …	4000.00	1

Example

- All records matching the condition will be deleted.
- Caution: As with **update**, if **where** is omitted, all records will be deleted. Sometimes you are warned…

```
99    -- unsafe
100 ⚠ DELETE FROM athlete;
```

Unsafe query: 'Delete' statement without 'where' clears all data in the table Execute Execute and Suppress

Check

Tasks – Modelling repetition

Imagine that you take up a position as an assistant in an institute at the University of Applied Sciences in Aachen.

Your first task is to reorganise the data of the experiments, as your predecessor has been dismissed due to his chaotic working style and the disappearance of experimental data.
You are dismayed to discover that files containing experimental data and parameters for various experiments are scattered around the server in no apparent order. You want to get a first overview with a database and store all found information.

1. Model a design in an ER diagram according to the following requirements.

   Please do not have a look at the following tasks yet, just read the requirements and design the ER diagram!

Check

Tasks – Modelling repetition – Requirements

- A table for experiments, in which you can store information about the experiment and which you can also assign files to, seems useful to you.
- For each experiment there is an arbitrary describing text as well as the information when this experiment was last worked on.

  Example: Experiment 'Cold Fusion', last edited on '1.11.2014, 12:02:03'.
- For the files belonging to the experiment, you decide that in this first version you do not want to store the content itself, but only the path to an existing file. It is also useful to be able to store a description of the contents of the file.
- As you wish to handle both data and parameter files, store the type of file in a type: a '1' stands for a parameter file, a '2' for a data file.

  Example: The parameter file (type '1'), located at 'c:/params/P1', contains 'temperature and pressure'. This file belongs to the Cold Fusion experiment.

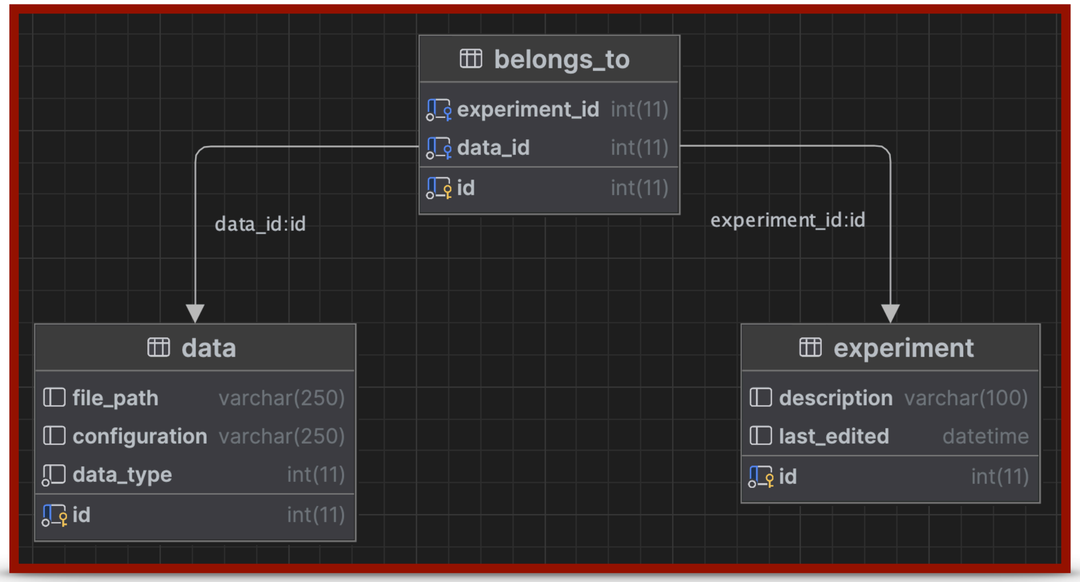
Check

Tasks – SQL commands

2. A design is shown opposite.
   Your ER diagram should look something like this.

   Create a schema ami_experiment and corresponding tables for the entity types experiment, data and belongs_to.

   Write the SQL statements on paper first for practice, and then test them in your DBMS.



Check

Tasks – SQL commands

3. Add the following data to the tables using SQL commands.

   Start on paper here as well (at least for a few insert commands).

id	file_path	configuration	data_type
33	c:/params/P1	{T=1.3e8, P=1.4e6}	1
34	c:/params/P2	{v=0.5c}	1
35	c:/params/P3	{T=1.6e8, P=1.2e6}	1
42	c:/daten/D1a	{1/2}	2
43	c:/daten/D1b	{2/2}	2
44	c:/daten/D2	{1/1}	2

data

id	experiment_id	data_id
1	15	33
2	15	34
3	15	42
4	15	43
5	16	34
6	16	35
7	16	44

belongs_to

id	description	last_edited
15	Cold Fusion	2014-11-01 12:02:03
16	Hot Fusion	2014-11-02 14:05:06
17	Explosive Gas	2014-10-03 17:08:09

experiment

Check

Tasks

4. Now another attempt is made to run the 'Cold Fusion' experiment and new data is received or changed.

   Change the date of the last access of the experiment 'Cold Fusion' to 5. Nov 2014, 18:09 and 10 sec.
   Attention: In this problem you have to find the `id` of the experiment 'Cold Fusion' by using a `subselection` and you are not allowed to use the condition `id=15` directly!

5. `Select` the data sets available for this 'Cold Fusion' experiment.
   As a reminder: Data sets are of type 2, while parameter sets are of type 1.

id	file_path	configuration	data_type
42	c:/daten/D1a	{1/2}	2
43	c:/daten/D1b	{2/2}	2

Result

Check

Tasks

6. Use the results of task 5 to generate slightly modified new data in **data**, but without individual **insert** statements. This is an **insert** based on multiple records.
   For once, assume that the new **ids** differ from the old ones by 3. No auto-increment should be used. Append '**_V2**' to the **file path**.

id	file_path	configuration	data_type
45	c:/daten/D1a_V2	{1/2}	2
46	c:/daten/D1b_V2	{2/2}	2

7. Analogous to task 6, add the appropriate entities to **belongs_to**, so that the data from task 6 also belongs to the 'Cold Fusion' experiment.
   For the required **ids** in **belongs_to**, choose an offset of your choice, e.g. +5, as described above.
   The new type 2 data in **belongs_to** looks like this.

Results

id	experiment_id	data_id
8	15	45
9	15	46

Check

Tasks

8. Find all the files belonging to the 'Cold Fusion' experiment and generate an output in this form:

id	file_path	configuration
33	c:/params/P1	{T=1.3e8, P=1.4e6}
34	c:/params/P2	{v=0.5c}
42	c:/daten/D1a	{1/2}
43	c:/daten/D1b	{2/2}
45	c:/daten/D1a_V2	{1/2}
46	c:/daten/D1b_V2	{2/2}

Result

9. Since an error was found in this experiment with `id=15`, `delete` all the corresponding data in `data` and `belongs_to`. The experiment itself remains.
10. Remove the schema `ami_experiment`.

Unit 0x08
Trigger, Stored Procedures

Trigger, Stored Procedures

Goal

- See Stored Procedures and Triggers in action - just a quick overview.

Remarks

- In order to be able to pass a definition of a stored procedure or a trigger to the DBMS, the meaning of the command terminator, the semicolon ';', has to be temporarily changed, because otherwise a ';' within the procedure or the trigger would lead to the end of the definition itself. Therefore, you define your own replacement before the definition with `delimiter`, use the ';' within the definition of the procedure or trigger as normal, end the whole definition with the new replacement and set the command terminator back to ';' at the end. See examples.
- Often '//' or '$$' are used as replacements.

Stored Procedures

Intention

- Creating, calling and deleting stored procedures.

Example

- The following **create procedure** statement creates a 'Stored Procedure' called **ShowProducts** that queries the product table.
- The procedure is called with **call**,
- and deleted, in the same way as tables, with **drop**.

```
DELIMITER //
CREATE PROCEDURE ShowProducts()
BEGIN
  SELECT P.* FROM shop_product P;
END //
DELIMITER ;

CALL ShowProducts();

DROP PROCEDURE ShowProducts;
```

db_unit_0x08_training

- Note: In **DataGrip**, the entire block must be selected and executed so that the command delimiters are set correctly.

Stored Procedures

Intention

- Create stored procedures with parameters.

Example

- First, a procedure called ShowOneProduct is defined, which is passed an id parameter of type int to filter the selection.
- Outlook: There are not only stored procedures, but also stored functions that can return results. There are also other parameter types, but that is not relevant here.

```
DELIMITER //
CREATE PROCEDURE ShowOneProduct(IN id int)
BEGIN
  SELECT P.* FROM shop_product P where P.id=id;
END //
DELIMITER ;

CALL ShowOneProduct( id: 11); -- 'Meatballs'
```

db_unit_0x08_training

Stored Procedures

Intention

- Understanding triggers in action.

Example

- This create trigger command creates a before_product_update trigger that is called before a product update and makes an entry in a log table for each change, containing the old and new values of the attribute name and a timestamp.

```
CREATE TABLE IF NOT EXISTS log_update (
  id INT(11) NOT NULL AUTO_INCREMENT,
  table_name VARCHAR(250) NULL DEFAULT NULL,
  last_text VARCHAR(250) NULL DEFAULT NULL,
  new_text VARCHAR(250) NULL DEFAULT NULL,
  last_update DATETIME NULL DEFAULT NULL,
  PRIMARY KEY (id)
);
```

db_unit_0x08_training

```
DELIMITER $$
CREATE TRIGGER before_product_update BEFORE UPDATE ON shop_product
FOR EACH ROW BEGIN
    INSERT INTO log_update (table_name,last_text,new_text,last_update)
    VALUES ('shop_product',OLD.name,NEW.name,NOW());
END$$
DELIMITER ;
```

```
UPDATE shop_product set name='Buletten' where id=11;
UPDATE shop_product set name='Meatballs' where id=11;
```

```
SELECT * FROM log_update;
```

id	table_name	last_text	new_text	last_update
1	shop_product	Meatballs	Buletten	2024-01-05 07:37:15
2	shop_product	Buletten	Meatballs	2024-01-05 07:37:17

Check

Tasks

1. Create a stored procedure (with or without parameters) of your choice, run it and then delete it.

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Databases
SQL – You made it!

Prof. Dr. rer. nat. Alexander Voß
Inform-Professur

FH Aachen
University of applied Sciences