SQL Basics

Learn SQL the easy way

Fabian Gaussling

Contents

Preface

More than 40 years ago SQL was invented. SQL, that's the acronym for Structured Query Language and the concepts behind it are more relevant than ever. Nowadays a lot of data is generated and collected in all areas. Therefor it is important to handle those data correctly. For that SQL is still necessary. The concept of relational databases exists for several decades and one could not imagine life without it.

Nearly every application saves data in a database. Also a lot of hardware components like sensors, engines, etc. generate data. Based on this data totally new use cases for reporting occurred, like improvement of productivity, quality control, etc. Because of that SQL is getting more interesting for people from IT distant areas: engineers, medical staff, etc.

This book should be an introduction for everyone who is interested in SQL. The book starts with a chapter on theoretical concepts and basic terms followed by an overview on the course's data model. Chapter three deals with the creation and alteration of database objects (creating, changing and dropping tables, columns, constraints, etc.) as well as with the manipulation of data (insert, update, delete).

Chapter four to six are about basic query techniques (selecting columns, doing calculations, filtering data), joining multiple tables (inner and outer joins) and grouping & aggregating data. This is the foundation of every SQL query and when you've accomplished chapter 6 you are able to get most of your problems solved with the help of SQL.

The seventh chapter is on sub queries, a more advanced technique, which is needed for complex calculations when one single query isn't enough. Mostly that are questions that you could also solve with multiple standalone queries but in order to reduce manual work you can combine these standalone queries to a big query with many sub-queries. Chapter eight is dealing with set operations like union, intersection and exception.

The last chapter is a collection of multiple smaller topics like granting/revoking rights, smaller useful functions and an introduction to analytical functions.

Before you get started you should set up a suitable working environment for the exercises. For that you can download a guide on my homepage: http://gaussling.com/information/book_ressources_en You find information where to find the database software (Oracle, MySQL, Postgres, SQL Server) and how to install and configure the training database.

Now you can start directly into the world of databases and SQL.

Have fun reading this book!

Fabian Gaussling

1 Introduction to database systems

What is a database?

Almost everyone is familiar with Excel and has used it at least once in their lifetime. In Excel, you have worksheets consisting of many cells where you can enter different values (numbers, text, dates, etc.). These cells are organized in columns and rows.

For instance, if you wanted to manage addresses in Excel, you would probably enter the different data in different columns, e.g.:

- First name
- Surname
- Street
- ZIP code
- City

You could arrange the address data in rows, and the whole thing would have the following format:

First name	Surname	Street	ZIP code	City
Hugo	Schmidt	Sylter Weg 15	24145	Kiel
Bert	Meier	Schanzenstraße 1	20357	Hamburg
Egon	Müller	Exerzierplatz 3	24103	Kiel
Ivonne	Müller	Oldendorfer Weg 22	25524	Itzehoe

This constitutes an address table. We also have tables in databases, the so-called ***database tables***. Every table consists of several ***columns***, each containing different data. The rows are called datasets in database terminology. A ***dataset*** contains all columns that belong together. In our example, a dataset would consist all the columns of an address. Database tables contain different forms of data depending on the application.

In practice, you usually need more than one table (just like in Excel). You could have one table for customers, one for sold products, and another for the billings. The three tables contain very different data, but all three tables could be important for the order management of a pizza delivery service. When you combine several tables, we speak of a ***database***. Different databases for diverse applications are managed in a ***database management system (DBMS)***. Common examples include Oracle, Microsoft SQL Server, IBM DB2, etc.

One uses the so-called query language for easy access of the data in the tables. The query language enables you to tell the database the datasets that you need. Assuming you wanted to view Hugo's address from our previous example, you can use the ***Structured Query Language (SQL)***. A simple example would look like this:

```
SELECT *
FROM TBL_ADDRESSES
WHERE Surname='Schmidt'
```

This would return all columns of the dataset that contain the surname 'Schmidt'. Such a language is particularly advantageous if you have tables containing millions of datasets or if you want to view specific data from multiple tables at the same time (will be described in detail later). With that, we have successfully used our very first SQL in this course. Many more will follow. ;)

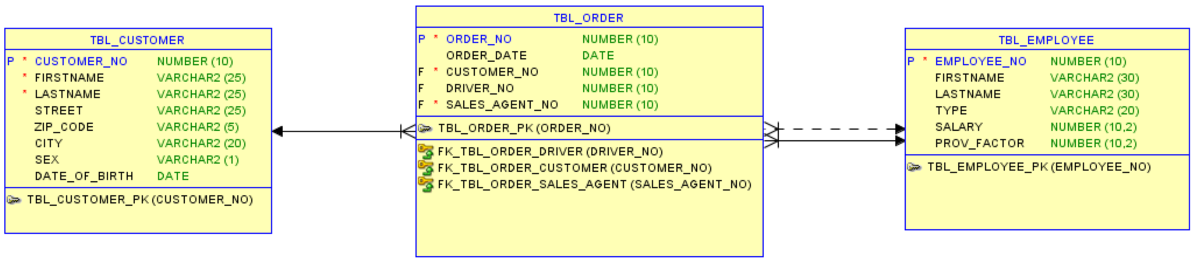
In addition to SQL as a query language, we also have the ***Data Definition Language (DDL)*** and ***Data Manipulation Language (DML)***. The DDL is used to define the structure of a database, i.e. tables and their structures. Furthermore, a database also has numerous other objects (e.g., views, indices, etc.), which will only be touched on marginally in this course. The DML is used to fill, edit, or delete data in tables.

Relationships between tables

Most databases contain diverse tables in order to structure the data in an organized manner and avoid redundancies (see chapter on normalization).

The different tables are hereby **related** to each other. For example, if you have one table containing information about your customers and another for the orders, you would only need to add the customer numbers in the orders table and not the entire information about the customer. Having this customer number in both tables creates a relationship between these two tables. All the information about a customer (e.g., address, phone no., etc.) is only saved and managed once in the customers table instead of saving it multiple times in the database. For example, you can use SQL to determine how many orders have been placed by customers from a specific region by simply linking the customers table (containing the customers' place of residence) and the orders table. If a customer changes his/her address, you only need to update the table at a single point.

We use ***ER models*** (ER = entity relationship) to illustrate the relationship between tables via their columns. Below is an example of such a model:



TBL_CUSTOMER
P * CUSTOMER_NO NUMBER (10)
* FIRSTNAME VARCHAR2 (25)
* LASTNAME VARCHAR2 (25)
STREET VARCHAR2 (25)
ZIP_CODE VARCHAR2 (5)
CITY VARCHAR2 (20)
SEX VARCHAR2 (1)
DATE_OF_BIRTH DATE
TBL_CUSTOMER_PK (CUSTOMER_NO)
TBL_ORDER
P * ORDER_NO NUMBER (10)
ORDER_DATE DATE
F * CUSTOMER_NO NUMBER (10)
F DRIVER_NO NUMBER (10)
F * SALES_AGENT_NO NUMBER (10)
TBL_ORDER_PK (ORDER_NO)
FK_TBL_ORDER_DRIVER (DRIVER_NO)
FK_TBL_ORDER_CUSTOMER (CUSTOMER_NO)
FK_TBL_ORDER_SALES_AGENT (SALES_AGENT_NO)
TBL_EMPLOYEE
P * EMPLOYEE_NO NUMBER (10)
FIRSTNAME VARCHAR2 (30)
LASTNAME VARCHAR2 (30)
TYPE VARCHAR2 (20)
SALARY NUMBER (10,2)
PROV_FACTOR NUMBER (10,2)
TBL_EMPLOYEE_PK (EMPLOYEE_NO)

Figure 1: A simple ER model

The relationships or links between tables are established via so-called keys. Here, a distinction is made between primary and foreign keys. ***Primary keys*** uniquely identify a dataset in a table. In the case of the customers table, this would be the 'customer no.'. This is marked with a "P" in the ER model to highlight the primary key. A ***foreign key*** is a reference to the primary key of another table (marked with "F" in the ER model), i.e. which dataset in the other table is linked to that of the current table. In the above example: Which customer dataset (primary key) is linked to which order (via the foreign key). Relationships are therefore always defined by linking a foreign key to a primary key.

There are different types of relationships in different tables:

1. 1:1 – relation
2. 1:n – relation

3. N:m – relation
4. Recursive relations

In the ***1:1 relation***, you only have one dataset in the foreign table for every dataset in the primary table and vice versa. As for the ***1:n relation***, you have 1 to n datasets in the primary table for every dataset in the foreign table. And for every dataset in the primary table, you only have 1 dataset in the foreign table. This is the most common case in practice. In our example, we have a 1:n relationship between customers and orders. Every customer can appear several times in the orders table. However, there only exists one customer for every order. ***N:m relationships*** mean that for every dataset in one table, you have n datasets in the other table. In relational databases, such a situation can only be modeled using an intermediate table.

Most DBMS allow you to ensure that the ***referential integrity*** is upheld. This means that every foreign key must always have a corresponding dataset in the referenced table. In such a case, it would not be possible to delete this dataset from the foreign table as long as it is referenced by another dataset. You can only delete it after you delete all the other datasets first. The same also applies in reverse. You must first create this dataset in the foreign table before referencing it from another table.

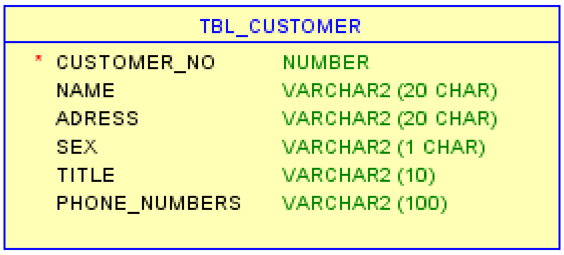
For you to always ensure this referential integrity, you must set up the so-called ***foreign key constraints***. This simply means defining the foreign-key relationships between two tables at a database level. We also have the so-called ***check constraints***. These are used to ensure that only certain values can be entered in specific columns such as if the column is only meant for the salutations 'Mr., Mrs., & Ms.'.

Normal forms

The following properties must always be ensured for easy and correct data evaluation:

- non-redundancy
- unique
- consistency

To make sure these properties are upheld, there are certain rules that must be adhered to in data models. These rules are referred to as ***normal forms***. There are five normal forms, each of which can be used to avoid specific redundancies that can arise in the data model. The first three normal forms are usually the most relevant and are described in detail using the contact information table below:

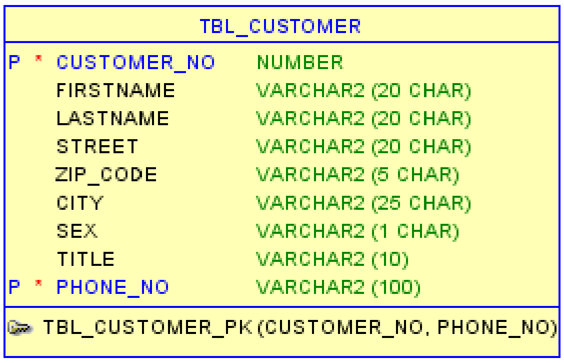


TBL_CUSTOMER		
*	CUSTOMER_NO	NUMBER
	NAME	VARCHAR2 (20 CHAR)
	ADRESS	VARCHAR2 (20 CHAR)
	SEX	VARCHAR2 (1 CHAR)
	TITLE	VARCHAR2 (10)
	PHONE_NUMBERS	VARCHAR2 (100)

Figure 2: Example of customer data – starting point

A table in the ***1st normal form*** cannot have repetition groups and each of its attributes must be atomic (i.e. cannot be subdivided into other attributes).

Applied to our example, this means that we cannot have "Address" as an attribute since the address can be subdivided into street, ZIP code, and city (atomicity). If a contact can have several phone numbers, these should not be saved in the “Phone no.” field (repetition group). We also cannot have fields “Phone_number 1” … “Phone_number X” (repetition group). Instead, we could add an attribute called PHONE_NUMBER as part of the key (from a combination of the CUSTOMER_NUMBER and PHONE_NUMBER).



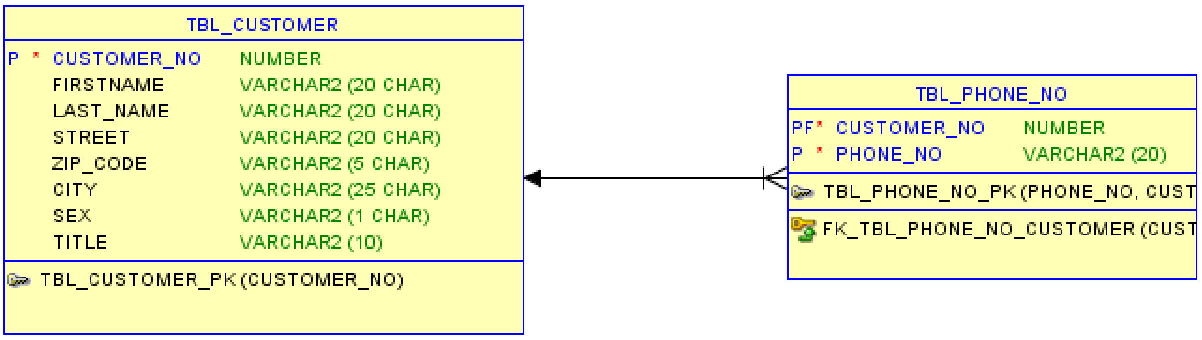
```
TBL_CUSTOMER
P * CUSTOMER_NO     NUMBER
    FIRSTNAME       VARCHAR2 (20 CHAR)
    LASTNAME        VARCHAR2 (20 CHAR)
    STREET          VARCHAR2 (20 CHAR)
    ZIP_CODE        VARCHAR2 (5 CHAR)
    CITY            VARCHAR2 (25 CHAR)
    SEX             VARCHAR2 (1 CHAR)
    TITLE           VARCHAR2 (10)
P * PHONE_NO        VARCHAR2 (100)
TBL_CUSTOMER_PK (CUSTOMER_NO, PHONE_NO)
```

Figure 3: Example of customer data – 1st normal form

The 1st NF basically makes sure that the data can be evaluated. If you were to save the entire address in a single attribute, it would be difficult, e.g., to sort and filter the data based on the place of residence.

The ***2nd normal form*** stipulates that given the table is in the 1st NF, every non-key field is dependent on the entire key and not just a part of the key. This ensures that a table only contains relational data. It also helps avoid inconsistencies since the attributes can only occur once.

In our example, this means that NAME, STREET, CITY, GENDER, and SALUTATION depend only on the CUSTOMER_NUMBER but not the TELEPHONE_NUMBER. If a contact was to have several phone numbers, the contact information would be redundant in the table thus consuming too much space (no longer a big problem today) and can also lead to inconsistencies. You would now create an extra table for phone numbers.



```
TBL_CUSTOMER
P * CUSTOMER_NO     NUMBER
    FIRSTNAME       VARCHAR2 (20 CHAR)
    LAST_NAME       VARCHAR2 (20 CHAR)
    STREET          VARCHAR2 (20 CHAR)
    ZIP_CODE        VARCHAR2 (5 CHAR)
    CITY            VARCHAR2 (25 CHAR)
    SEX             VARCHAR2 (1 CHAR)
    TITLE           VARCHAR2 (10)
TBL_CUSTOMER_PK (CUSTOMER_NO)

TBL_PHONE_NO
PF* CUSTOMER_NO     NUMBER
P * PHONE_NO        VARCHAR2 (20)
TBL_PHONE_NO_PK (PHONE_NO, CUST
FK_TBL_PHONE_NO_CUSTOMER (CUST
```

Figure 4: Example of customer data – 2nd normal form

To comply with the ***3rd normal form***, a table must already be in the 2nd NF and non-key fields must be independent of other non-key fields. The 3rd NF prevents further redundancies.

In our example, this would mean that fields ‘City’ and ‘Salutation’ would be outside the attribute since the city is dependent on the ZIP code and the salutation dependent on the gender. This additionally helps avoid redundancies that would lead to inconsistencies.

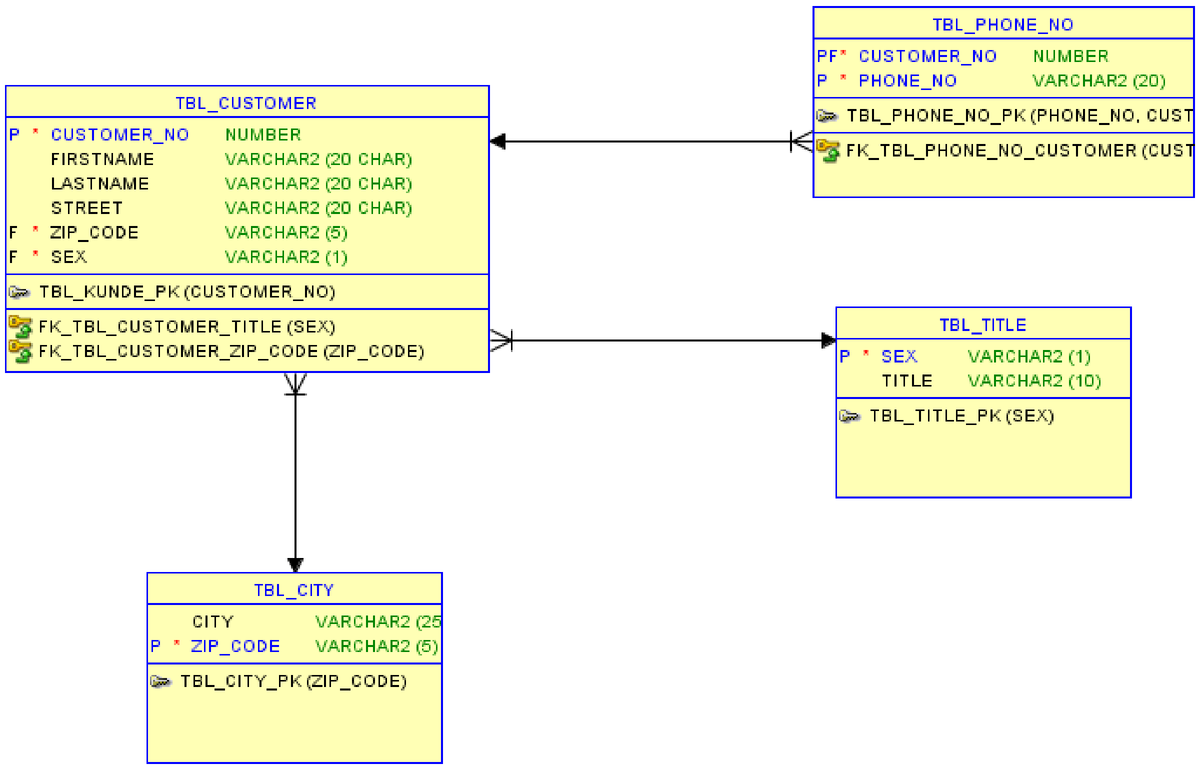

TBL_CUSTOMER
P * CUSTOMER_NO NUMBER
FIRSTNAME VARCHAR2 (20 CHAR)
LASTNAME VARCHAR2 (20 CHAR)
STREET VARCHAR2 (20 CHAR)
F * ZIP_CODE VARCHAR2 (5)
F * SEX VARCHAR2 (1)
TBL_KUNDE_PK (CUSTOMER_NO)
FK_TBL_CUSTOMER_TITLE (SEX)
FK_TBL_CUSTOMER_ZIP_CODE (ZIP_CODE)
TBL_PHONE_NO
PF* CUSTOMER_NO NUMBER
P * PHONE_NO VARCHAR2 (20)
TBL_PHONE_NO_PK (PHONE_NO, CUST
FK_TBL_PHONE_NO_CUSTOMER (CUST
TBL_TITLE
P * SEX VARCHAR2 (1)
TITLE VARCHAR2 (10)
TBL_TITLE_PK (SEX)
TBL_CITY
CITY VARCHAR2 (25
P * ZIP_CODE VARCHAR2 (5)
TBL_CITY_PK (ZIP_CODE)

Figure 5: Example of customer data – 3rd normal form

The normalization increases the number of tables in a data model. Thus, queries can take longer since the tables must be linked to each other. In case the response times become too long, you can also ***de-normalize*** the data models. This is usually common in data warehouse applications that have tables in the so-called star schema.

Database transactions

When adding or changing data in tables, it must be ensured that the data remain consistent. Here, one usually talks of the ***ACID*** paradigm:

- ***A***tomicity
- ***C***onsistency
- ***I***solation
- ***D***urability

The paradigm dictates that data changes must be ***atomic***, meaning that a set of operations is either performed completely or not at all. In addition, it must also be ascertained that after the changes, the dataset remains ***consistent*** (as long as it was consistent before). The changes should be ***isolated*** so that different operations do not to interfere with each other (e.g., simultaneous deletion and reading of a dataset). The edited data must be saved ***permanently***.

For all this to be upheld, we have the so-called ***transactions*** in modern DBMS. A transaction is more like a bracket with which one or more DML (insert, update, delete) instructions are handled as a block. The results are only saved after the individual instructions have been executed. After a transaction, the results can either be saved permanently using ***COMMIT*** or rolled back using ***ROLLBACK***. The changes are only visible after a successful COMMIT.

Exercise

1. Modify the following data structure to obtain the 3rd NF. The data are hereby arranged in columns:

Country	**ISO code**	**State**	**City**	**Branch**	**Employee**	**Contact data**
Germany	GER	Hamburg	Hamburg	Spitaler Straße	Mr. Schmidt Mrs. Müller Mr. Meyer …	Tel: 040-1234 Fax.: 040-1235 Email: Hamburg@enterprise.com
Denmark	DEN	n/a	Copenhagen	…	Mrs. Sörensen …	…
…						

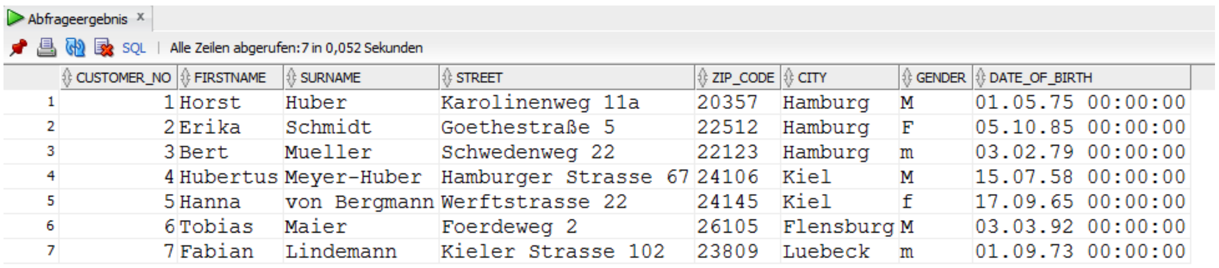
3 Creating simple queries

Select columns (SELECT)

In this chapter, we will create our first simple queries. We will only query data from a single table and gradually get to learn the basic structure of an SQL statement. The simplest example looks like this:

```
SELECT *
FROM TBL_CUSTOMERS
```

Result:



Abfrageergebnis ×

SQL | Alle Zeilen abgerufen: 7 in 0,052 Sekunden

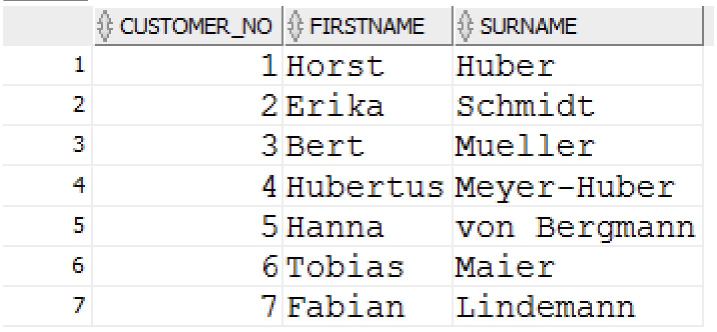
	CUSTOMER_NO	FIRSTNAME	SURNAME	STREET	ZIP_CODE	CITY	GENDER	DATE_OF_BIRTH
1	1	Horst	Huber	Karolinenweg 11a	20357	Hamburg	M	01.05.75 00:00:00
2	2	Erika	Schmidt	Goethestraße 5	22512	Hamburg	F	05.10.85 00:00:00
3	3	Bert	Mueller	Schwedenweg 22	22123	Hamburg	m	03.02.79 00:00:00
4	4	Hubertus	Meyer-Huber	Hamburger Strasse 67	24106	Kiel	M	15.07.58 00:00:00
5	5	Hanna	von Bergmann	Werftstrasse 22	24145	Kiel	f	17.09.65 00:00:00
6	6	Tobias	Maier	Foerdeweg 2	26105	Flensburg	M	03.03.92 00:00:00
7	7	Fabian	Lindemann	Kieler Strasse 102	23809	Luebeck	m	01.09.73 00:00:00

The asterisk stands for all columns in the table. FROM <Table name> defines the table from which the data should be selected. If we only want to view specific columns, we can specify this by entering the column name. We can also use commas to select multiple columns:

```
SELECT CUSTOMER_NO,
       FIRST_NAME,
       SURNAME
FROM TBL_CUSTOMERS
```

This returns the following three columns:

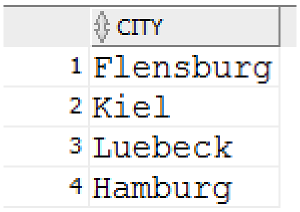
Result:



	CUSTOMER_NO	FIRSTNAME	SURNAME
1	1	Horst	Huber
2	2	Erika	Schmidt
3	3	Bert	Mueller
4	4	Hubertus	Meyer-Huber
5	5	Hanna	von Bergmann
6	6	Tobias	Maier
7	7	Fabian	Lindemann

If we now query the only column CITY, the result would be:

Result:

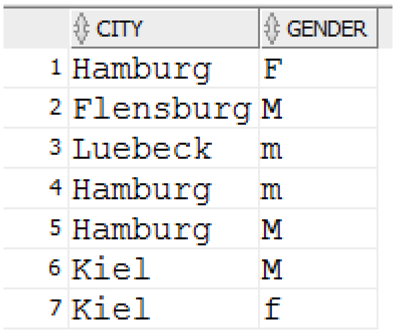


	CITY
1	Flensburg
2	Kiel
3	Luebeck
4	Hamburg

This returns the city for all datasets in the table. Using the keyword ***DISTINCT*** changes the result as follows:

```
SELECT DISTINCT CITY
FROM TBL_CUSTOMERS
```

Result:



	CITY	GENDER
1	Hamburg	F
2	Flensburg	M
3	Luebeck	m
4	Hamburg	m
5	Hamburg	M
6	Kiel	M
7	Kiel	f

As you can see, using DISTINCT removes all duplicates and the different cities are only listed once. Here, all columns are always considered. This means that if you use DISTINCT for CITY and GENDER, the different combinations from the two columns will be displayed once. Duplicates in the combinations of the different column values are thus excluded.

```
-- Selecting distinct combinations

SELECT DISTINCT CITY, GENDER
FROM TBL_CUSTOMERS
```

Result:

Here, you have 2 rows for Hamburg and Kiel respectively since two columns have been selected and the duplicates eliminated for both CITY and GENDER.

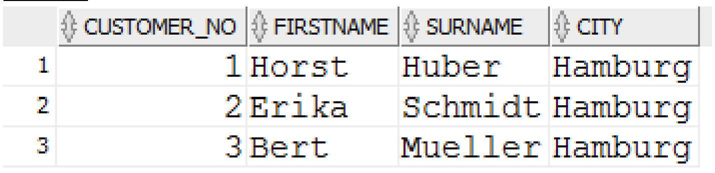
You can use a double minus '--' to add ***comments till the end of the line*** in SQL statements. If you want to ***comment in multiple lines*** you have to put the comment between /* ..... */ .

Selecting rows (WHERE)

The next component of an SQL statement that you can use is the WHERE clause. This makes it possible to define the so-called filters, whereby you can select specific rows:

```
SELECT CUSTOMER_NO, FIRST_NAME, CITY
FROM TBL_CUSTOMER
WHERE CITY='Hamburg'
```

Result:



	CUSTOMER_NO	FIRSTNAME	SURNAME	CITY
1	1	Horst	Huber	Hamburg
2	2	Erika	Schmidt	Hamburg
3	3	Bert	Mueller	Hamburg

This statement returns all rows with “Hamburg” as the city. In our example, this corresponds to the first three customers. There are different operators with which you can define filters:

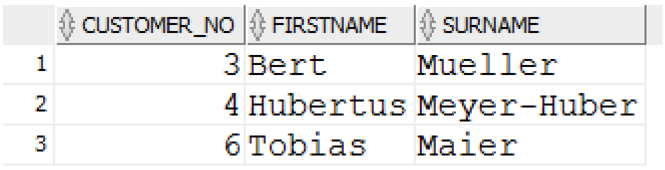
Operator	Function
=	Equal
<>	Unequal
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal
IS (NOT) NULL	Checks whether a column is (not) NULL
BETWEEN <value1> AND <value2>	Between <value1> and <value2>
IN (<value1>, <value2>, …)	Contained in (…)
LIKE ‘…..’	Text similarity. % can be used as a placeholder for multiple arbitrary characters. _ is used as a placeholder for exactly one character.
<Operator> ANY (<value1>,…)	<Operator> can be: =, <, >, <>, <=, >= Oracle translates it to: <Operator> <value1> OR <Operator> <value2> OR … In practice, it is mostly replaced by other comparisons and hence rather uncommon.
<Operator> ALL (<value1>,…)	<Operator> can be: =, <, >, <>, <=, >= Oracle translates it to: <Operator> <value1> AND <Operator> <value2> AND … In practice, it is mostly replaced by other comparisons and hence rather uncommon.

Table 2: SQL comparison operators

For example, if you want to view all customers whose surname starts with an M, you could use the following SQL statement:

```
SELECT CUSTOMER_NO, FIRST_NAME, SURNAME
FROM TBL_CUSTOMERS
WHERE SURNAME LIKE ‘M%’
```

Result:



	CUSTOMER_NO	FIRSTNAME	SURNAME
1	3	Bert	Mueller
2	4	Hubertus	Meyer-Huber
3	6	Tobias	Maier

‘M%’ means that the first letter in the string must be an M. The rest of the characters after that do not matter. This is achieved by the % sign. It is a placeholder for any number of characters. If you want a placeholder for exactly one character, you should use an underscore _.

Exercise 1: Display all products whose price is greater than 10€.

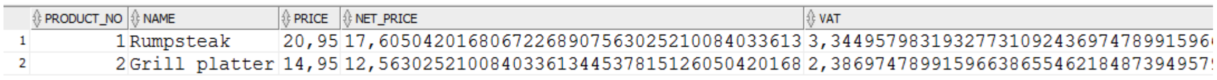
Exercise 2: Display all customers with their FIRST_NAME, SURNAME, and ZIP code if they come from ZIP code regions 24 + 25 (Tip: Do not use the LIKE operator).

Using calculations and functions

You can also perform calculations on the columns in the SELECT section and use them in filters as follows:

```
SELECT PRODUCT_NO, NAME, PRICE, PRICE/1.19 AS NET, PRICE/1.19*0.19 AS VAT
FROM TBL_PRODUCT
WHERE PRICE/1.19*0.19 > 2
```

Result:



	PRODUCT_NO	NAME	PRICE	NET_PRICE	VAT
1	1	Rumpsteak	20,95	17,60504201680672268907563025210084033613	3,3449579831932773109243697478991596
2	2	Grill platter	14,95	12,56302521008403361344537815126050420168	2,3869747899159663865546218487394957

As the example shows, you can also perform normal and well-known computational operations in SQL – both when using SELECT and the WHERE clause. In the above example, this has been used to only display rows with a VAT of more than 2 euros. You can use AS to rename columns. You can also use brackets.

The following math operators are hereby available:

Operator	Function
+, -, *, /	Addition, subtraction, multiplication, division with decimal places
mod (x,y)	Modulo division
^	Power

Table 3: Math operators

There are different functions that are implemented in databases. These can be used for rounding values, replacing characters in character strings, type conversion, computation with date values, etc. Below is a description of the most important functions:

Function/Syntax	Description
To_Date(<Value> [, <Format>])	Converts a string into a date. Here, you can specify a format string to define the structure of the value. For example, if you want to convert string ‘20120201’ into a date, you must specify the format string ‘YYYYMMDD’.

To_Char(<Value> [, <Format>])	Converts a date into a character string. You can specify a format string to define how the character string should appear. For example, you can easily convert date 01.02.2012 into '201202' using the format string 'YYYYMM'.
To_Number (<character string>)	Converts a character string into a number
Round (<number>, <figures>)	Rounds the number <number> commercially to <figures> figures
Substr (<Text>, <Start>, <no. of characters>)	Returns a text section from <Text>, starting from position <Start> and with <no. of characters> characters
Length (<Text>)	Returns the length of a character string in <Text>
InStr (<Text>, <character string>, <Start Pos.>)	Searches <character string> in <Text> starting from <Start Pos.> and returns the position
Replace (<Text>, <characters>, <new_characters>)	Replaces all <characters> characters with <new_characters> in <Text>
Concat (<Text1>, <Text2>, …)	Connects text 1 … n to a character string. Alternatively, you can also use the || operator.
LTrim/RTrim (<Text> [, <characters>])	Trims all <characters> characters to the left/right of <Text>. If no characters are specified, empty spaces are removed.
NVL (<field>, <null_value_characters>)	Replaces NULL values in <field> with <null_value_characters>
ADD_MONTHS (<date>, <months<)	Adds <months> months to <date> and returns the corresponding date
LAST_DAY (<date>)	Returns the last day of the month from <date>
UPPER / LOWER (<text>)	Converts all characters from <Text> to upper/lower case
LPad/RPad(<text>, <width> [, <characters>])	Fills string <text> with <characters> up to <width> characters. If no characters are specified, the string is filled with empty spaces.
ABS (<number>)	Returns the absolute number
SYSDATE / SYSTIMESTAMP	Returns the current system date/current system stamp (i.e. date and time)
TRUNC (<number>, <count>)	Truncates the <number> up to <count> decimal places. It is not rounded. If you do not specify the <count>, the number is cut off at the decimal point.

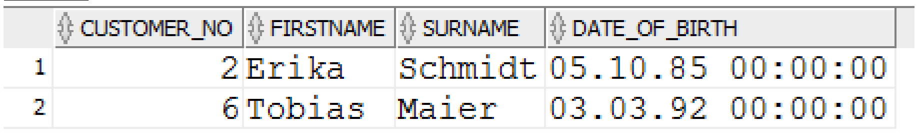
Table 4: Standard functions in Oracle

You can find a comprehensive overview of all possible format string components at: http://docs.oracle.com/cd/B28359_01/olap.111/b28126/dml_commands_1029.htm#OLADM780.

You can use the functions both in the SELECT as well as the WHERE section of an SQL statement, thus allowing you further filter options:

```
SELECT CUSTOMER_NO, FIRST_NAME, SURNAME, Date_of_Birth
FROM TBL_CUSTOMERS
WHERE to_number(  to_char(Date_of_Birth, ‘YYYY‘))>=1980
```

Result:

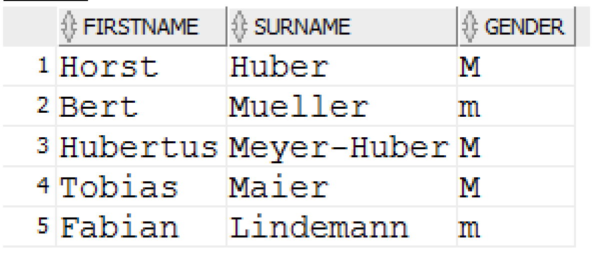


	CUSTOMER_NO	FIRSTNAME	SURNAME	DATE_OF_BIRTH
1	2	Erika	Schmidt	05.10.85 00:00:00
2	6	Tobias	Maier	03.03.92 00:00:00

As you can see, one can also combine multiple functions. This example selects all customers born after 1980. The string function can then be used to filter out all male customers:

```
SELECT FIRST_NAME, SURNAME, GENDER
FROM TBL_CUSTOMERS
WHERE UPPER(GENDER)=‘M‘
```

Result:



	FIRSTNAME	SURNAME	GENDER
1	Horst	Huber	M
2	Bert	Mueller	m
3	Hubertus	Meyer-Huber	M
4	Tobias	Maier	M
5	Fabian	Lindemann	m

Why use the UPPER function? An uppercase M and a lowercase m could have been used interchangeably for the gender. Alternatively, you can also do the same using the ‘IN’ operator: IN(‘M’, ‘m’).

Exercise 3: Select all customers who celebrate their birthday in the first quarter of a year.

Exercise 4: Use SQL functions to display all customers whose first name starts with an F (do not use the LIKE operator)!

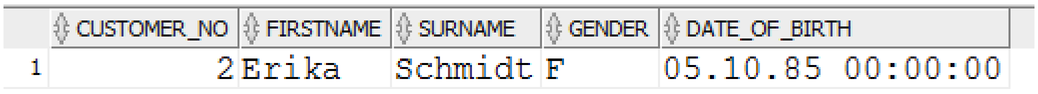
Exercise 5: Use SQL functions to select all customers whose surname end with ‘mann’ (do not use the LIKE operator). Tip: substr + length

Combining multiple filters

It is often necessary to select datasets based on multiple criteria. For example, you might want to view all female customers who were born after 1970:

```
SELECT CUSTOMER_NO, FIRST_NAME, SURNAME, GENDER, DATE_OF_BIRTH
FROM TBL_CUSTOMERS
WHERE to_number(  to_char(DATE_OF_BIRTH, 'YYYY'))>=1970
AND GENDER IN('F', 'f')
```

Result:



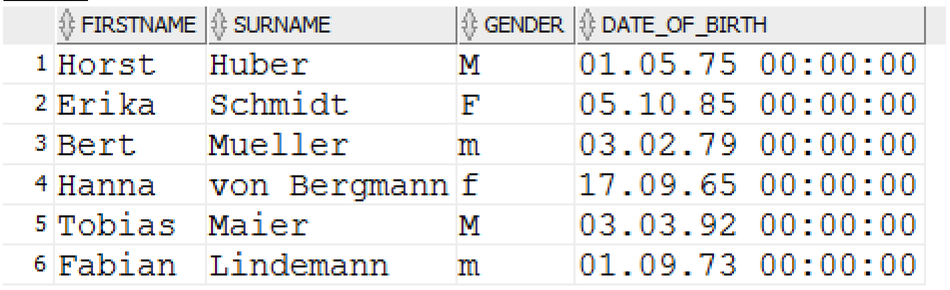
	CUSTOMER_NO	FIRSTNAME	SURNAME	GENDER	DATE_OF_BIRTH
1	2	Erika	Schmidt	F	05.10.85 00:00:00

You can combine multiple filters using ***AND***. This means that both conditions must be met for the required dataset to be returned. The second dataset is not displayed in this case since the Date_of_Birth does not meet the requirements.

You can also combine conditions using ***OR***. This means that the dataset is returned if A or B is met:

```
SELECT FIRST_NAME, SURNAME, GENDER, DATE_OF_BIRTH
FROM TBL_CUSTOMERS
WHERE GENDER IN('F', 'f')
OR  to_number( to_char(DATE_OF_BIRTH, 'YYYY'))>=1970
```

Result:

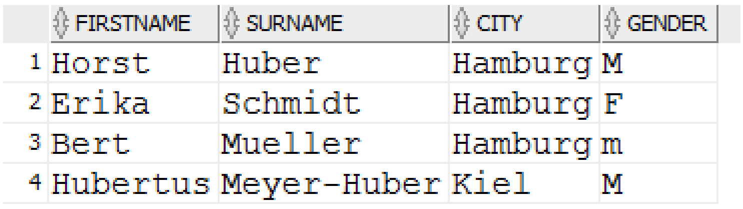


	FIRSTNAME	SURNAME	GENDER	DATE_OF_BIRTH
1	Horst	Huber	M	01.05.75 00:00:00
2	Erika	Schmidt	F	05.10.85 00:00:00
3	Bert	Mueller	m	03.02.79 00:00:00
4	Hanna	von Bergmann	f	17.09.65 00:00:00
5	Tobias	Maier	M	03.03.92 00:00:00
6	Fabian	Lindemann	m	01.09.73 00:00:00

This returns all customers who are female or customers who were born after 1970. The whole thing becomes more complex if you combine AND and OR. In this case, the AND operator takes precedence over the OR operator:

```
SELECT FIRST_NAME, SURNAME, CITY, GENDER
FROM TBL_CUSTOMERS
WHERE CITY='Hamburg' OR CITY='Kiel' AND GENDER IN('M', 'm')
```

Result:



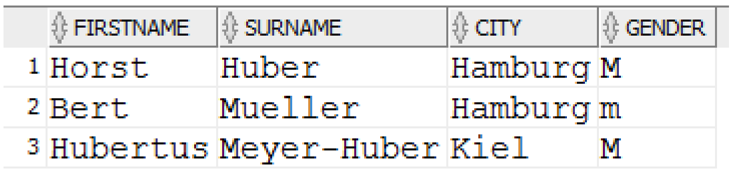
	FIRSTNAME	SURNAME	CITY	GENDER
1	Horst	Huber	Hamburg	M
2	Erika	Schmidt	Hamburg	F
3	Bert	Mueller	Hamburg	m
4	Hubertus	Meyer-Huber	Kiel	M

Sometimes you need to use brackets depending on what you want to evaluate. According to the statement defined above, the query returns customers who live in Hamburg (whether male or female) or male customers from Kiel.

If you want to view male customers who come from Kiel or Hamburg, you must extend the SQL as follows:

```
SELECT FIRST_NAME, SURNAME, CITY, GENDER
FROM TBL_CUSTOMERS
WHERE (CITY='Hamburg' OR CITY='Kiel') AND GENDER IN('M', 'm')
```

Result:



	FIRSTNAME	SURNAME	CITY	GENDER
1	Horst	Huber	Hamburg	M
2	Bert	Mueller	Hamburg	m
3	Hubertus	Meyer-Huber	Kiel	M

The dataset "Erika Schmidt" is now excluded since the datasets to be returned must now fulfill the following conditions:

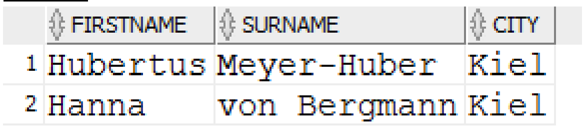
1. male
2. from Kiel or Hamburg

This was achieved by using brackets. The condition (CITY='Hamburg' OR CITY='KIEL') is evaluated first. The result is then analyzed to see which of the customers are male.

In the context of AND and OR, we also have the NOT operator. For example, this can be used to check which customers do not come from Kiel, Flensburg, or Lübeck:

```
SELECT FIRST_NAME, SURNAME, CITY
FROM TBL_CUSTOMERS
WHERE CITY NOT IN ('Hamburg', 'Flensburg', 'Lübeck')
```

Result:



	FIRSTNAME	SURNAME	CITY
1	Hubertus	Meyer-Huber	Kiel
2	Hanna	von Bergmann	Kiel

Exercise 6: Select all products with product group 1 and a price greater than 15€.

Exercise 7: Select all products with a VAT value <0.75€ or >2€ from product groups 1, 2, or 4. Make sure to display all columns of the product table and additionally display the VAT (VAT rate: 19%).

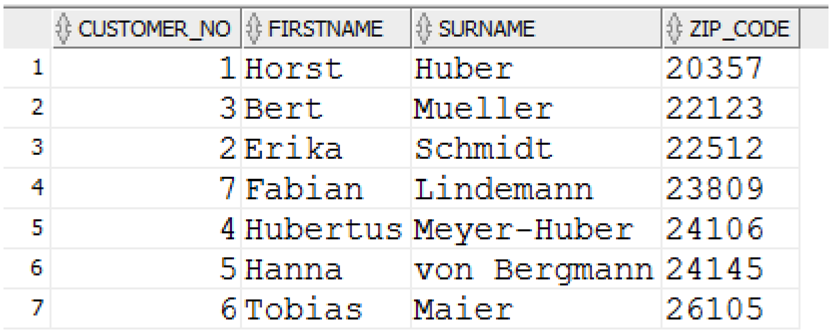
Sorting results

You can also use different criteria to sort the displays query results, e.g., based on the ZIP code:

```
SELECT CUSTOMER_NO, FIRST_NAME, SURNAME, ZIP_code
```

```
FROM TBL_CUSTOMERS
ORDER BY PLZ ASC
```

Result:

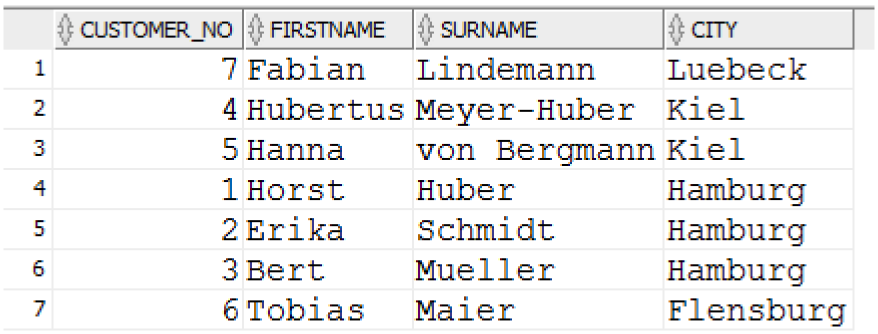


	CUSTOMER_NO	FIRSTNAME	SURNAME	ZIP_CODE
1	1	Horst	Huber	20357
2	3	Bert	Mueller	22123
3	2	Erika	Schmidt	22512
4	7	Fabian	Lindemann	23809
5	4	Hubertus	Meyer-Huber	24106
6	5	Hanna	von Bergmann	24145
7	6	Tobias	Maier	26105

Below is another example on sorting results:

```
SELECT CUSTOMER_NO, FIRST_NAME, SURNAME, CITY
FROM TBL_CUSTOMERS
ORDER BY 4 DESC, 1 ASC
```

Result:



	CUSTOMER_NO	FIRSTNAME	SURNAME	CITY
1	7	Fabian	Lindemann	Luebeck
2	4	Hubertus	Meyer-Huber	Kiel
3	5	Hanna	von Bergmann	Kiel
4	1	Horst	Huber	Hamburg
5	2	Erika	Schmidt	Hamburg
6	3	Bert	Mueller	Hamburg
7	6	Tobias	Maier	Flensburg

As you can see, you can use ***ASC*** or ***DESC*** to define the direction in which you want to sort the data. Sorting columns can be specified either by name or position. You can combine multiple sorting criteria using commas.

NULLS FIRST and ***NULLS LAST*** enables you to place zeroes at the start or end of a list during sorting. The rest is then sorted as described.

One brief topic to conclude this chapter: Oracle has a table called DUAL. This has exactly one dataset. This small table can be helpful at times since in Oracle, you must specify a table in the FROM clause in Oracle. For example, if I want to quickly try out a function, I can use the following statement:

```
SELECT SYSDATE
FROM dual;
```

This returns exactly one value, i.e. the system date in this case.

4 Querying multiple tables

We now know how to create simple queries in a table. However, most applications require access to multiple tables. Looking at the customers or products alone is somehow boring. We want to know where and how often the products were sold. For us to be able to do this, this chapter describes the different ‘Join’ types for relational databases.

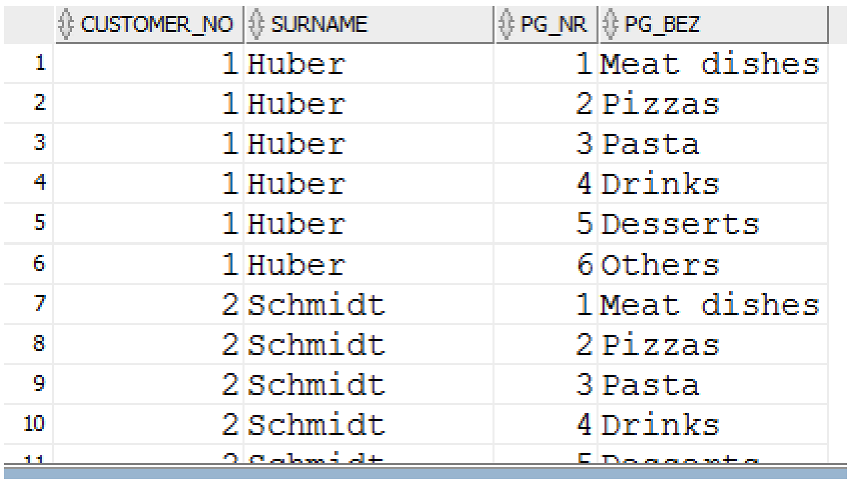
Cross product

The simplest way to query data from multiple tables is by simply writing a second table in the FROM clause. However, in most cases, this does not return what you want. This is because, in this case, relational databases combine all datasets from table A with all datasets from table B. For large tables, this can result in very large outputs that can push your database performance to its limits.

However, you also have cases where cross products can be useful. Let us first look at the syntax for a cross product:

```
SELECT
        TBL_CUSTOMERS.CUSTOMER_NO,
        TBL_CUSTOMERS.SURNAME,
        TBL_PRODUCT_GROUP.PRODUCT_GROUP_NO AS PG_NO,
        TBL_PRODUCT_GROUP.NAME AS PG_NAME
FROM TBL_CUSTOMERS, TBL_PRODUCT_GROUP
ORDER BY 1,3
```

<u>Result:</u>

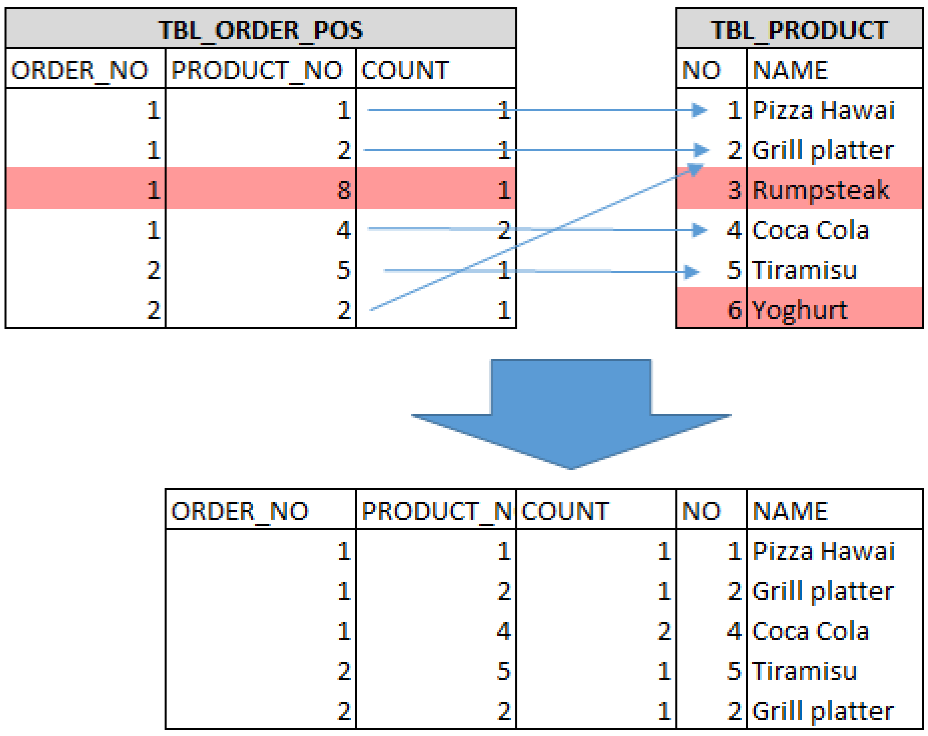


	CUSTOMER_NO	SURNAME	PG_NR	PG_BEZ
1	1	Huber	1	Meat dishes
2	1	Huber	2	Pizzas
3	1	Huber	3	Pasta
4	1	Huber	4	Drinks
5	1	Huber	5	Desserts
6	1	Huber	6	Others
7	2	Schmidt	1	Meat dishes
8	2	Schmidt	2	Pizzas
9	2	Schmidt	3	Pasta
10	2	Schmidt	4	Drinks
11	2	Schmidt	5	Desserts

As you have seen, you only need to specify the tables that you want to define a cross product for in the FROM clause. You can also do the same in SELECT and specify which columns should be taken from which table using the following syntax: ***<Table name> . <Column name>***
The above example also shows the effect of the cross product. We have exactly seven datasets in TBL_CUSTOMERS, five in TBL_PRODUCT_GROUP. In total, each dataset from TBL_CUSTOMERS will be combined with every dataset from TBL_PRODUCT_GROUP, resulting in 35 datasets.

Inner joins

Contrary to the cross product, inner joins links the two tables via one or more columns. This is illustrated in the example of the inner join between products and order positions:



TBL_ORDER_POS		
ORDER_NO	PRODUCT_NO	COUNT
1	1	1
1	2	1
1	8	1
1	4	2
2	5	1
2	2	1

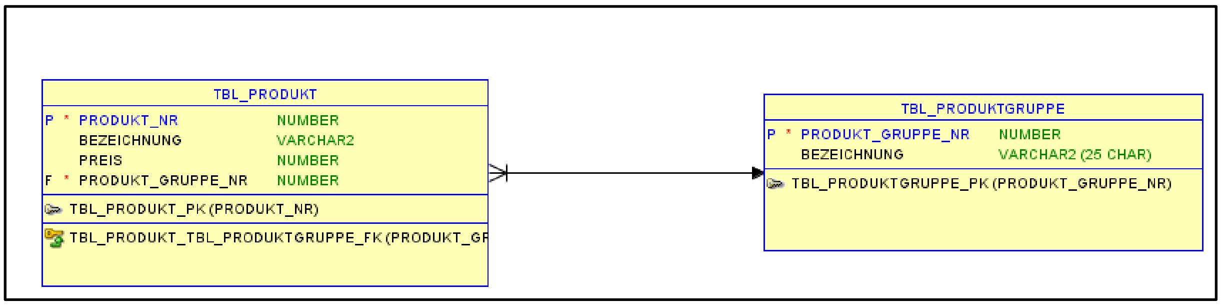
TBL_PRODUCT	
NO	NAME
1	Pizza Hawai
2	Grill platter
3	Rumpsteak
4	Coca Cola
5	Tiramisu
6	Yoghurt

ORDER_NO	PRODUCT_N	COUNT	NO	NAME
1	1	1	1	Pizza Hawai
1	2	1	2	Grill platter
1	4	2	4	Coca Cola
2	5	1	5	Tiramisu
2	2	1	2	Grill platter

Figure 11: Inner join between TBL_PRODUCT and TBL_ORDER_POS

The two tables should be linked via the PRODUCT_NO. To do this, the database searches for datasets in which the values are identical in the link columns. These two datasets are then merged to get the resulting set. The links are indicated by the thin blue arrows. The red arrows have no correspondence in the other table and are therefore excluded from the resulting datasets.

Let us look at another simple example and assume we want to know the product group to which our products belong. Here, we have the product group number in the product table. However, this number is not exactly meaningful for most users and it would be better to also display the name of the product group from the product group table. For us to do this, we must link TBL_PRODUCT with TBL_PRODUCT_GROUP. We can do this via the PRODUCT_GROUP_NO column since, according to the data model, this column provides a link between the two tables.



```
TBL_PRODUKT
P * PRODUKT_NR            NUMBER
    BEZEICHNUNG           VARCHAR2
    PREIS                 NUMBER
F * PRODUKT_GRUPPE_NR     NUMBER
TBL_PRODUKT_PK (PRODUKT_NR)
TBL_PRODUKT_TBL_PRODUKTGRUPPE_FK (PRODUKT_GF
```

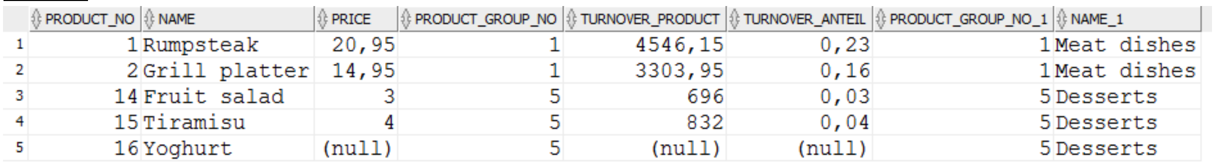
```
TBL_PRODUKTGRUPPE
P * PRODUKT_GRUPPE_NR     NUMBER
    BEZEICHNUNG           VARCHAR2 (25 CHAR)
TBL_PRODUKTGRUPPE_PK (PRODUKT_GRUPPE_NR)
```

Figure 12: Section of product and product group from the data model

The following SQL creates a link between the two tables and returns all columns:

```
SELECT *
FROM TBL_PRODUCT prd
         JOIN TBL_PRODUCT_GROUP  grp
         ON prd.PRODUCT_GROUP_NO=grp.PRODUCT_GROUP_NO
WHERE grp.PRODUCT_GROUP_NO IN(1,5)
```

Result:



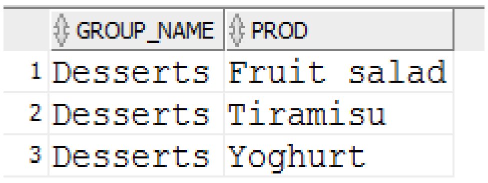
	PRODUCT_NO	NAME	PRICE	PRODUCT_GROUP_NO	TURNOVER_PRODUCT	TURNOVER_ANTEIL	PRODUCT_GROUP_NO_1	NAME_1
1	1	Rumpsteak	20,95	1	4546,15	0,23	1	Meat dishes
2	2	Grill platter	14,95	1	3303,95	0,16	1	Meat dishes
3	14	Fruit salad	3	5	696	0,03	5	Desserts
4	15	Tiramisu	4	5	832	0,04	5	Desserts
5	16	Yoghurt	(null)	5	(null)	(null)	5	Desserts

This lets us view the product groups 1 and 5 with their products. The link is established using ***<Table 1> JOIN <Table 2> ON <Conditions>***. Prd and grp are called table aliases. They allow us to give the tables different, simpler names and then access the columns. In the above example, the DBMS first goes through all columns of the product table while checking the product_group_no in each row. The database then uses this number to link to the product_group table and selects all datasets with this number. The fields of the dataset from TBL_PRODUCT and TBL_PRODUCT_GROUP are added to a new, resulting dataset. This is done for each row in the product table. In the end, only the datasets that belong to product group 1 or 5 are displayed. If you do not limit the fields, all fields from both tables are taken. For fields that have the same name (e.g., PRODUCT_GROUP_NO), the DBMS automatically adds a number at the end to make them unique.

In the case of an inner join, only the datasets that have a corresponding dataset in both table A and table B are returned. Datasets that do not have a referenced dataset in table B are excluded from the result. This is illustrated in the example below. There is no product assigned to product group 6 (=others):

```
SELECT grp.NAME AS GROUP, prd.NAME AS PROD
FROM TBL_PRODUCT prd
         JOIN TBL_PRODUCT_GROUP grp
         ON prd.PRODUCT_GROUP_NO=grp.PRODUCT_GROUP_NO
WHERE grp.PRODUCT_GROUP_NO IN(5,6)
```

Result:

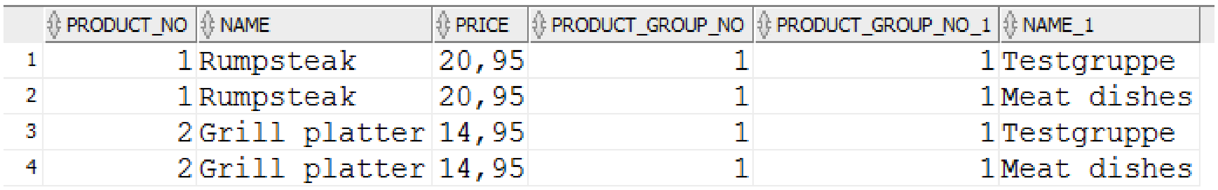


	GROUP_NAME	PROD
1	Desserts	Fruit salad
2	Desserts	Tiramisu
3	Desserts	Yoghurt

Let's now look at a small effect with joins and use it to manipulate the data in the product group table:

```
ALTER TABLE TBL_PRODUCT DISABLE CONSTRAINT FK_PROD_PROD_GROUP;

ALTER TABLE TBL_PRODUCT_GROUP DISABLE CONSTRAINT PK_PROD_GROUP;

INSERT INTO TBL_PRODUCT_GROUP VALUES (1, ‘Test group’);

COMMIT;

SELECT *
FROM TBL_PRODUCT prd
        JOIN TBL_PRODUCT_GROUP grp
        ON prd.PRODUCT_GROUP_NO=grp.PRODUCT_GROUP_NO
WHERE grp.PRODUCT_GROUP_NO=1;
```

Result:



	PRODUCT_NO	NAME	PRICE	PRODUCT_GROUP_NO	PRODUCT_GROUP_NO_1	NAME_1
1	1	Rumpsteak	20,95	1	1	Testgruppe
2	1	Rumpsteak	20,95	1	1	Meat dishes
3	2	Grill platter	14,95	1	1	Testgruppe
4	2	Grill platter	14,95	1	1	Meat dishes

What just happened here? There are two entries for the same PRODUCT_GROUP_NO in the product group table, i.e. 1 meat dishes and 1 test group: With join, all datasets from TBL_PRODUCT_GROUP are taken if the join condition is met. Since both groups have 1 as the number, they both fulfill the criterion and are therefore returned. Thus, the database combines every product from group no. 1 with both groups and hence result in duplicates. This is a very common phenomenon, especially if you have deactivated foreign key constraints (which we have already covered with the ALTER commands).

Next, we will use the following brief script to restore to the former state:

```
DELETE TBL_PRODUCT_GROUP WHERE NAME=‘Test group’;

COMMIT;

ALTER TABLE TBL_PRODUCT_GROUP ENABLE CONSTRAINT PK_PROD_GROUP;

ALTER TABLE TBL_PRODUCT ENABLE CONSTRAINT FK_PROD_PROD_GROUP;
```

Exercise 1: Create a query that shows you the name of the customers who placed an order in February 2013.

You can also create queries for more than two tables. This basically works the same way as with just two tables. The only difference is that each additional table must be added using a separate join in the query.

Exercise 2: Which drivers (no., name) delivered to customer ‘Müller’ in April 2013?

Exercise 3: Which products (name, price) were purchased by customers from Kiel in the 2nd quarter of 2013?

Sometimes, you also need to create joins over more than one column. The additional join conditions (join columns) are linked using the AND operator in the ON clause.

About the author

I am working as a freelancing database developer, trainer and author. I am living in Kiel, Germany. For more than 10 years I am already working in this area. My customers are companies of different sizes from different industries (banking, engineering, medical, etc.).

Besides my regular work I am writing books and articles for my blog on database topics like SQL, NoSQL, MDX, Cognos, datawarehousing, etc. True to my motto „practical experiences for practical applications" a lot of practical tutorials and guidebooks are created in that process in order to help you finding your way through the database jungle.

More information about me: born in 1981, MSc in information management, experiences as customer and consultant.

Do you have further questions?

Just contact me via eMail: fabian@gaussling.com

Or get connected with me on…

… XING

… LinkedIn

… Google+

… Twitter

If you want more database knowledge, you can find it ….

… on my BLOG: http://bi-solutions.gaussling.com

… in my online database training: http://online-trainings.gaussling.com