Learning SQL the Badass Way

Edgar Treischl

4/29/23

Table of contents

Preface

Learning SQL the badass way is work in progress, keep that in mind if you start reading on these pages.

Learning SQL the *badass way* shows you the basic commands to manage tables and data with SQL. I assume that you are familiar with R (or other programming languages) to manipulate data. For this reason, this page only summarizes main idiosyncrasies of SQL, not concepts that you probably know from any other programming experience.

I made the experience that learning new things always takes a lot of time and often I have the impression that many courses are designed without taking your personal experience into account. What a surprise, I can read and write some SQL code even I have no idea what SQL does and I guess you can too! This book is the result of this experience and gives a quick and dirty introduction, assuming that you have coding experience and I do not have to explain why we want to wrangle data in the first place.

If you do not have this impression or if you want to learn SQL in a proper way, close this book. Go and find your way to a real SQL course or some other resources. However, if you are bored from long introduction what data is, why we need to learn how to wrangle data or other common aspects that come along the data science journey, feel free to join my SQL journey.

As we already know, learning SQL is beneficial but maybe we need more motivational input. ChatGTP gives us the following reasons why we should learn SQL even if you are fluent in a language such as R:

- *Efficient data management*: SQL is designed to work with relational databases, which are ideal for managing large amounts of structured data. By learning SQL, R users can efficiently query, retrieve and manage data from databases using SQL commands, making their data analysis tasks more efficient.
- *Collaborative work*: SQL is a common language used by data analysts and data engineers, making it easier to collaborate and share data between different teams. By learning SQL, R users can communicate more effectively with other data professionals and work collaboratively on projects.
- *Integration with R*: R users often work with data that is stored in databases, and SQL provides a way to query and retrieve data from these databases directly into R. This integration allows R users to take advantage of the power of SQL for data management, while still working with their preferred R environment.

- *Advanced data manipulation*: SQL provides powerful features for data manipulation, including filtering, sorting, aggregating and joining data from multiple tables. By learning SQL, R users can take advantage of these advanced features to manipulate their data in more sophisticated ways.
- *Job market demand*: SQL is a widely used skill in the data analytics job market. By learning SQL, R users can broaden their skillset and increase their job market competitiveness.

In addition, ChatGTP also helps us to get an overview about the most important R packages that are commonly used for working with SQL, including:

- `dplyr`: A powerful package for data manipulation that can connect to various SQL databases and perform operations such as filtering, grouping, and joining (Wickham et al. 2023).
- `DBI`: An R package that provides a common interface for connecting to various SQL databases (, Wickham, and Müller 2022).
- `RMySQL` and `RSQLite`: Packages that provide an interface for connecting to MySQL and SQLite databases, respectively (Ooms et al. 2022; Müller et al. 2023).
- `RJDBC`: A package that provides a JDBC interface for connecting to various databases, including Oracle, Microsoft SQL Server, and PostgreSQL (Urbanek 2022).
- `RODBC`: A package that provides an ODBC interface for connecting to various databases, including Microsoft SQL Server and PostgreSQL (Ripley and Lapsley 2022).
- `sqldf`: A package that allows you to run SQL queries on data frames in R (Grothendieck 2017).

Overall, the choice of which package to use will depend on the specific database you're working with and your preferred interface. However, these packages should provide a good starting point for working with SQL in R, but first we need to outline some SQL basics.

1 Introduction

The biggest mistake trying to learn SQL first. If you ever opened a book that introduces SQL (or any other programming language) to will find several chapters that outline you what SQL (structured query language) is. An explanation from the Wikipedia page should be sufficient: "SQL a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS)". Thus, in a nutshell we can remember that SQL is a language to manage data (tables) in databases. The next chapters shows you how to setup the corresponding software and the database. I'll not do that in this blog because setting up a database on your own computer does not make much sense because there are easier ways for the first steps in SQL especially if you are a R user.

In case you do not have access to a SQL database: This page was built with `RMarkdown` and includes some SQL snippets with the `DBI` package. All you have to do is install the package, setup the connection and include your SQL code directly in your RMarkdown document or your R script. Thus, in our case there is no database installed, we can make use of the the local memory to simulate a database, save a table (data frame) and run SQL commands directly as a code chunk via our R script. Checkout the databases page from RStudio for more information. Furthermore, I hope you are familiar with the `mtcars` and the `iris` data if you want to reproduce these first SQL steps. Both are implemented in R, but it does not matter if you not familiar with these data sets. I picked them because a lot of people know them and this blog shows you the first basic SQL commands how retrieve and work with data.

1.1 Select

- Use select to retrieve a table or a column from a table
- You can select a single column from a table
- Or select the entire table (data frame) with the wildcard `*`

```
SELECT * FROM mtcars;
```

Table 1.1: Displaying records 1 - 10

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

- Limit the output by providing the number of lines

```sql
SELECT mpg, disp FROM mtcars LIMIT 5;
```

Table 1.2: 5 records

mpg	disp
21.0	160
21.0	160
22.8	108
21.4	258
18.7	360

- You can also insert a starting point that skips some observations. For instance, `OFFSET 10` will skip the first ten table entries
- You must use quotations marks if the column contains special characters (like `'Petal.Width'` from iris data)

1.2 Where

- Define what you want to select with the `Where` option (SQL folks say clause)
- For instance, the variable `am` is a binary indicator (0/1) and you can use *where* to select data only if `am = 0`

```sql
SELECT * FROM mtcars WHERE am = 0 LIMIT 5;
```

Table 1.3: 5 records

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2

- Remember to use quotation marks if you try to use *where* with non-numerical values from: e.g. `!= 'label'`

```
SELECT * FROM iris WHERE Species = "virginica"  LIMIT 5;
```

Table 1.4: 5 records

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
6.3	3.3	6.0	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3.0	5.9	2.1	virginica
6.3	2.9	5.6	1.8	virginica
6.5	3.0	5.8	2.2	virginica

1.3 Count

- Count counts cases!

```
SELECT COUNT(*) FROM mtcars;
```

Table 1.5: 1 records

COUNT(*)
32

- We can count subgroups with the `WHERE` clause:

```
SELECT COUNT(am) FROM mtcars WHERE am != 0;
```

Table 1.6: 1 records

COUNT(am)
13

1.4 DISTINCT

- `Distinct` can be used to find distinct values. For instance, there are three different Species in the iris data, and distinct helps you to distract them:

```
SELECT DISTINCT Species FROM iris
```

Table 1.7: 3 records

Species
setosa
versicolor
virginica

- As in other programming languages, we can combine several commands. For instance, we can `COUNT` how many `Distinct` species the iris data has:

```
SELECT COUNT (DISTINCT Species) FROM iris
```

Table 1.8: 1 records

COUNT (DISTINCT Species)
3

1.5 Insert Values

- Next, I use a small data set (df) to illustrate how to *insert values*, *make updates*, and *delete cases*
- My toy data set `df` has two observations with three variables: `x,y,z`
- Never mind if you do not know what a `tribble` is, it is just a command to create data

```
library(tidyverse)

df <- tribble(
  ~x, ~y,  ~z,
  "a", 2,  3.6,
  "b", 1,  8.5
)
df
```

```
# A tibble: 2 x 3
  x         y     z
  <chr> <dbl> <dbl>
1 a         2   3.6
2 b         1   8.5
```

- Now, we can insert new values into `df` by providing a list of the columns you want to fill in with values for each column:

```
INSERT INTO df (x, y, z) VALUES('c', 3, 1);
```

Let's see whether it worked:

```
SELECT * FROM df;
```

Table 1.9: 3 records

x	y	z
a	2	3.6
b	1	8.5
c	3	1.0

1.6 Updates

- Make updates for single (or multiple) values
- For instance, we can update the variable z and set `z = 77` for a certain level of another variable:

```
UPDATE df SET z = 77 WHERE x ='b';
```

- Take care, without the WHERE clause all observation would get the new value!

```
SELECT * FROM df;
```

Table 1.10: 3 records

x	y	z
a	2	3.6
b	1	77.0
c	3	1.0

1.7 Delete

- We can drop or delete observations, but of course we should take care since we probably do not want to delete the entire table, just for some implausible values
- For this reason we use the WHERE clause again, for instance, to get rid of second row of the toy data set:

```
DELETE from df WHERE x = 'b';
```

```
SELECT * FROM df;
```

Table 1.11: 2 records

x	y	z
a	2	3.6
c	3	1.0

In summary, in SQL we have to select a table from the database, specify conditions with the where clause. We can use count and distinct to get a first impression of the data. Furthermore, it is all about table. Sometimes we have to insert values, a really vague concept if you are use to work with data, but from a SQL you give your database a update, you are really right in front of the data or imagine that stream of new data needs an update. If that something you have to get used to it, don't be afraid, me too.

2 Data management

This is the second blog post from "Learn SQL the badass way". I outlined the scope of the blog in the first blog entry where I explained why I set the scope to R users and people with some programming languages. You may not find what you are looking for if you are not familiar with R or if you do not have any other experience to work with data, because I focus on the data management part in this blog.

Before we get in touch with new SQL commands, we have to learn some SQL vocabulary. As in other languages, we have to learn some basic vocabulary to advance our SQL skills. Let's say we have a small collection of tables about books. In the `author` table we store information about the book `author`(e.g. first name, second name, etc); in the `book` table contains information about each book (e.g. genre etc); and the `sales` table summarizes the sales data for each book. How do we manage all of these tables and the dependencies in SQL?

Sometimes you encounter a diagram to display how each table is related to each other and we can think of all the independent tables as a collection of tables and we have to figure out how they are related. A entity relationship diagram (ERD) will help you to see the relation of each table, it displays the collection of entities and attributes.

In the SQL world, *entities* are independent objects. For instance, the book table is a independent object because it exits next to other entities of our collection. Entities have *attributes* or properties. For instance, the book table contains title, subtitle, book id and more attributes. Thus, entities refer to tables (or in my world data frame) of our collection and attributes refer to columns, or I would say variables, in the table.

Furthermore, we can differentiate between primary and foreign keys in the tables:

- *Primary key*: Is a unique indicator that helps us to match tables (e.g. a unique author ID)
- *Foreign keys*: Is a primary key that is defined in other table to create a link between tables (Book ID in Book table and the sales table)

We may also differentiate what information a attribute stores. Some common data formats are:

- Char (for characters)
- Varchar (for variable character length)
- Numeric (Integer, time)

2.1 Types of SQL Statements

In the SQL world, we can ultimately distinguish between the data definition and manipulation language:

- *Data Definition Language (DDL)*:
  - Commands from the DDL are used to define, change, or drop tables (database)
  - SQL examples: Create, Alter, Truncate, Drop
- *Data Manipulation Language (DML)*:
  - DML is used to read and modify data in tables
  - Those operations are sometimes named as CRUD operations and we learned them in the last blog: *Create, Read, Update, and Delete* rows in a table
  - SQL examples: INSERT, SELECT, UPDATE, DELETE

Now, let's put some of these concepts into practice:

2.2 CREATE

- You can create new tables with `CREATE TABLE` command. It works in three steps. You have to provide a name for your table, each column needs a name, and you have to specify which kind of information will be stored (e.g. numerical values, characters) in the column
- The following command creates a toy table with for petsales with five variables:

```
CREATE TABLE PETSALE (
    ID INTEGER NOT NULL,
    PET CHAR(20),
    SALEPRICE DECIMAL(6,2),
    PROFIT DECIMAL(6,2),
    SALEDATE DATE
    );
```

As the output illustrates, we can add options to create the table:

- The `ID` variable is an integer that does not accept zero, or in other words: `NOT NULL`
- The column PET is generated to store to character string
- The column SALEDATE stores dates
- And we could also set a primary key with the clause: `PRIMARY KEY`
- A second example

```
CREATE TABLE PET (
    ID INTEGER NOT NULL,
    ANIMAL VARCHAR(20),
    QUANTITY INTEGER
    );
```

- So far, both tables do not contain any values. With `INSERT INTO`, we fill the table with corresponding values:

```
INSERT INTO PETSALE VALUES
    (1,'Cat',450.09,100.47,'2018-05-29'),
    (2,'Dog',666.66,150.76,'2018-06-01'),
    (3,'Parrot',50.00,8.9,'2018-06-04'),
    (4,'Hamster',60.60,12,'2018-06-11'),
    (5,'Goldfish',48.48,3.5,'2018-06-14');
```

- And for the second table:

```
INSERT INTO PET VALUES
    (1,'Cat',3),
    (2,'Dog',4),
    (3,'Hamster',2);
```

As we learned in the last session, we can use `SELECT` to check whether it worked:

```
SELECT * FROM PET;
```

Table 2.1: 3 records

ID	ANIMAL	QUANTITY
1	Cat	3
2	Dog	4
3	Hamster	2

2.3 ALTER TABLE

- We use the `ALTER TABLE` statement to add, delete, or modify columns. For instance:
- `ADD COLUMN, DROP COLUMN; ALTER COLUMN; RENAME COLUMN`

First: `ADD COLUMN`

```
ALTER TABLE PETSALE
ADD COLUMN QUANTITY INTEGER;
```

```
SELECT * FROM PETSALE;
```

Table 2.2: 5 records

ID	PET	SALEPRICE	PROFIT	SALEDATE	QUANTITY
1	Cat	450.09	100.47	2018-05-29	NA
2	Dog	666.66	150.76	2018-06-01	NA
3	Parrot	50.00	8.90	2018-06-04	NA
4	Hamster	60.60	12.00	2018-06-11	NA
5	Goldfish	48.48	3.50	2018-06-14	NA

- Again, fill in your values

```
UPDATE PETSALE SET QUANTITY = 9 WHERE ID = 1;
```

```
UPDATE PETSALE SET QUANTITY = 24 WHERE ID = 5;
```

- Check whether it worked

```
SELECT * FROM PETSALE;
```

Table 2.3: 5 records

ID	PET	SALEPRICE	PROFIT	SALEDATE	QUANTITY
1	Cat	450.09	100.47	2018-05-29	9
2	Dog	666.66	150.76	2018-06-01	NA
3	Parrot	50.00	8.90	2018-06-04	NA
4	Hamster	60.60	12.00	2018-06-11	NA
5	Goldfish	48.48	3.50	2018-06-14	24

Second: `DROP COLUMN`

```
ALTER TABLE PETSALE
DROP COLUMN PROFIT;
```

```
SELECT * FROM PETSALE;
```

Table 2.4: 5 records

ID	PET	SALEPRICE	SALEDATE	QUANTITY
1	Cat	450.09	2018-05-29	9
2	Dog	666.66	2018-06-01	NA
3	Parrot	50.00	2018-06-04	NA
4	Hamster	60.60	2018-06-11	NA
5	Goldfish	48.48	2018-06-14	24

Third: `ALTER COLUMN`

- We can change the data type, for instance, to increase the length of a character variable to VARCHAR(20) with `ALTER COLUMN`

```
ALTER TABLE PETSALE
ALTER COLUMN PET SET DATA TYPE VARCHAR(20);
```

```
SELECT * FROM PETSALE;
```

Table 2.5: 5 records

ID	PET	SALEPRICE	SALEDATE	QUANTITY
1	Cat	450.09	2018-05-29	9
2	Dog	666.66	2018-06-01	NA
3	Parrot	50.00	2018-06-04	NA
4	Hamster	60.60	2018-06-11	NA
5	Goldfish	48.48	2018-06-14	24

Forth: `RENAME COLUMN`

- Use `RENAME COLUMN` to change *to* a new name:

```
ALTER TABLE PETSALE
RENAME COLUMN PET TO ANIMAL;
```

```
SELECT * FROM PETSALE;
```

Table 2.6: 5 records

ID	ANIMAL	SALEPRICE	SALEDATE	QUANTITY
1	Cat	450.09	2018-05-29	9
2	Dog	666.66	2018-06-01	NA
3	Parrot	50.00	2018-06-04	NA
4	Hamster	60.60	2018-06-11	NA
5	Goldfish	48.48	2018-06-14	24

2.4 Truncate

- The `TRUNCATE` statement will remove all(!) rows from an existing table, just like the one we created in the beginning, however, it does not delete the table itself.

```
TRUNCATE TABLE PET IMMEDIATE;
```

Caution: `DROP TABLE tablename;` drops the entire table!

```
DROP TABLE PETSALE;
```

3 Calculations

Do you really want to “learn SQL the badass way”? I outlined the scope of the blog in the first blog entry where I explained why I set the scope to R users and people with some programming languages. You may not find what you are looking for if you are not familiar with R or if you do not have any other experience to work with data, because I focus on the data management part in this blog. Thus, I hope you find some helpful ressources to learn SQL, but I focus on data wrangling aspects, without explaining basic concepts to handle data.

In this session we learn how to use string patterns and ranges to search data. We will learn how to sort and group data to display result. Moreover, we practice composing nested queries and execute select statements to access data from multiple tables. For this reason I created a simple table that contains some attributes (y, z, id) about countries:

```
SELECT * FROM df ;
```

Table 3.1: 4 records

country	y	z	id
Germany	2	3.6	1
Austria	1	8.5	2
Brazil	4	2.5	3
Brazil	3	3.5	3

3.1 String values, ranges and set of values

We can use strings and the WHERE clause to search for specific observations. For instance, `WHERE countryname LIKE 'A%'` means that we search for country name column that start with the corresponding string. And we can use `%` as a wildcard character:

```
SELECT * FROM df WHERE country LIKE 'A%';
```

Table 3.2: 1 records

country	y	z	id
Austria	1	8.5	2

Use a range to select entries that depending on some criteria (> and <). In SQL, we specify `WHERE values are between 100 and 200`. Keep in mind that values are inclusive within the range. For instance, we can use the mtcars dataset and restrict the table with cars that have a horsepower (hp) between 100 and 200, we can even use an `AND` to restrict to cars with a manual transmission (AM = 1)

```
select * from mtcars
where (hp BETWEEN 100 and 200) and AM = 1 ;
```

Table 3.3: 5 records

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Another option gives us the `IN` operator. We can select columns by providing a list and the `IN` operator. As the next example shows, we select only those observations that match the provided list of the `IN` operator:

```
SELECT * FROM df WHERE country IN ('Brazil');
```

Table 3.4: 2 records

country	y	z	id
Brazil	4	2.5	3
Brazil	3	3.5	3

3.1.1 Sorting Result Sets

Sometimes we need to sort the entries alphabetically and we can do that with the `ORDER BY` clause. For instance, `ORDER BY country`:

```
SELECT * FROM df ORDER BY country;
```

Table 3.5: 4 records

country	y	z	id
Austria	1	8.5	2
Brazil	4	2.5	3
Brazil	3	3.5	3
Germany	2	3.6	1

- By default, the entries are ordered in an ascending order, but we can sort in a descending with `DESC` option as well:

```
SELECT * FROM df ORDER BY country DESC;
```

Table 3.6: 4 records

country	y	z	id
Germany	2	3.6	1
Brazil	4	2.5	3
Brazil	3	3.5	3
Austria	1	8.5	2

Sometimes we have several observations per unit or any kind of structural order, which is why we may want to order a specific variable. We can sort the data by providing the number of the column we want to use a sort. As the next example shows, we can use y (or 2) to sort the data:

```
SELECT * FROM df ORDER BY y;
```

Table 3.7: 4 records

country	y	z	id
Austria	1	8.5	2
Germany	2	3.6	1
Brazil	3	3.5	3
Brazil	4	2.5	3

3.1.2 Grouping result sets

To work with data, we have to get rid of duplicates and often it is much more easier if we restrict result set (data frame). To exclude duplicates we can use the `distinct()` command, which returns only distinct countries in our example:

```
SELECT distinct(country) FROM df ;
```

Table 3.8: 3 records

country
Germany
Austria
Brazil

In a similar fashion, maybe we have to clarify how many observations do we have per group? Or in our case, how many entries come from the same country and how often appears each level? In such a case we can count the county column and use the `group by` clause:

```
SELECT country, count (country) from df group by country;
```

Table 3.9: 3 records

country	count (country)
Austria	1
Brazil	2
Germany	1

As the last output showed, the count functions literally counts, but SQL does not give it a name, it simply displays what it does. We can change this ugly behaviour by providing a variable name and tell SQL how the column should be listed.

```
SELECT country, count (country) AS Count_Variable from df group by country;
```

Table 3.10: 3 records

country	Count_Variable
Austria	1

country	Count_Variable
Brazil	2
Germany	1

Certainly, counting is not the only function. We can estimate the mean average with `AVG()`. And now the average, little SQL monkey!

```
SELECT country, AVG(z) as Mean from df group by country;
```

Table 3.11: 3 records

country	Mean
Austria	8.5
Brazil	3.0
Germany	3.6

We can set a further conditions with a grouped by clause and add the `HAVING` option. As the next output shows, the `group by country HAVING count (country) > 1` returns only countries with more than one observation counted:

```
SELECT country, count (country) AS Count from df group by country having count (country) >
```

Table 3.12: 1 records

country	Count
Brazil	2

Let us try to remember that the `WHERE`clause is for entire result set; while `HAVING` works only for the `GROUPED BY` clause.

Congratulations! At this point you are able to:

- Use the `WHERE` clause to refine your query results
- Use the wildcard character (`%`) as a substitute for unknown characters in a pattern
- Use `BETWEEN` … `AND` to specify a range of numbers
- We can sort query results into ascending or descending order, by using the `ORDER BY` clause
- And we can group query results by using the `GROUP BY` clause.

3.2 Built-in Database Functions

We saw in the last section that we can aggregate (count, avg) data and use column functions. Most of the basic statistics functions (sum, avg, min, max) are available and we can specify further conditions, for instance, if we want to summarize groups:

```
select sum(mpg) as sum_mpg from mtcars where hp > 100
```

Table 3.13: 1 records

sum_mpg
401.4

Or we may use the scalar function and round to the nearest integer:

```
select round(drat, 1) as round_drat from mtcars
```

Table 3.14: Displaying records 1 - 10

round_drat
3.9
3.9
3.9
3.1
3.2
2.8
3.2
3.7
3.9
3.9

In SQL, there is a class of scalar functions. For instance, we can calculate the length of a string:

```
select length(country) from df
```

Table 3.15: 4 records

length(country)
7
7
6
6

Depending the SQL database you use, in db2 you can use the upper (UCASE) and lower case (LCASE) function for strings.

```
select upper(country) from df
```

Table 3.16: 4 records

upper(country)
GERMANY
AUSTRIA
BRAZIL
BRAZIL

In case of Oracle the functions are called lower and upper.

3.2.1 Date and Time Built-in Functions

Talking about SQL databases, there are three different possibilities to work with date and time DB2.
- Date: *YYYYMMDD* (Year/Month/Day) - Time: *HHMMSS* (Hours/Min/Sec) - Timestamp: *YYYYMMDDHHMMSSZZZZZZ* (Date/Time/Microseconds)

Depending on what you are up to do, there are functions to extract the day, month, day of month, day of week, day of year, week, hour, minute, and second. You can also extract the `current_date` and the `current_time`. Unfortunately, this does not work in Oracle the same way as in DB2, but to give you an example how to extract the day:

```
select day(date) from df where country = 'Germany'
```

3.2.2 Sub-Queries and Nested Selects

Sub-queries or sub selects are like regular queries but placed within parentheses and nested inside another query. This allows you to form more powerful queries than would have been otherwise possible. An example:

```
select avg(mpg) from mtcars
```

Table 3.17: 1 records

avg(mpg)
20.09062

Let's say we want to select only the observations with higher values than the average of mpg:

```
select * from mtcars where mpg > avg(mpg)
```

This would produce the following error: `misuse of aggregate function avg() Failed to execute SQL chunk`. One of the limitations of built in aggregate functions, like avg(), is that they cannot be evaluated in the WHERE clause always. Thus, we have to use sub-queries.

```
select * from mtcars where mpg >
  (select avg(mpg) from mtcars);
```

Table 3.18: Displaying records 1 - 10

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1

Colum expressions help to set sub queries as a list of columns. Say we select variable Z:

```
select country, z from df
```

Table 3.19: 4 records

country	z
Germany	3.6
Austria	8.5
Brazil	2.5
Brazil	3.5

And in the next step we add the average of all countries:

```
select country, z, avg(z) as avg_Z from df
```

Table 3.20: 1 records

country	z	avg_Z
Germany	3.6	4.525

This is obviously wrong. We cannot calculate on micro and macro level the same time, but we could use a sub-query (also called table expressions) to achieve it:

```
select country, z, (select avg(z) from df group by country) as avg_Z from df
```

Table 3.21: 4 records

country	z	avg_Z
Germany	3.6	8.5
Austria	8.5	8.5
Brazil	2.5	8.5
Brazil	3.5	8.5

Sub-queries can also be applied in the from clause. They are called derived tables or table expressions, because the outer query uses the results of the sub-query as a data source

```
select * from (select hp, vs from mtcars);
```

Table 3.22: Displaying records 1 - 10

hp	vs
110	0
110	0
93	1
110	1
175	0
105	1
245	0
62	1
95	1
123	1

3.2.3 Working with Multiple Tables

There are several ways to access multiple tables in the same query. Namely, using sub-queries, implicit join, and join operators, such as `INNER JOIN` and `OUTER JOIN`. For instance:

```
select * from df2;
```

Table 3.23: 2 records

country	valid	id
Germany	1	1
Austria	0	2

Let's say we want only observations from df that are listed in df2. In such a situation we can use a sub-queries:

```
select * from df
  where country in
  (select country from df2)
```

Table 3.24: 2 records

country	y	z	id
Germany	2	3.6	1

country	y	z	id
Austria	1	8.5	2

Of course, you could add also information of the second table and include only countries with a certain value:

```
select * from df
  where country in
  (select country from df2 where valid = 1)
```

Table 3.25: 1 records

country	y	z	id
Germany	2	3.6	1

Implicit joins implies that we can access multiple tables by specifying them in the `FROM` clause of the query. This leads to a CROSS JOIN (also known as Cartesian Join).

```
select * from df, df2
```

Table 3.26: 8 records

country	y	z	id	country	valid	id
Germany	2	3.6	1	Germany	1	1
Germany	2	3.6	1	Austria	0	2
Austria	1	8.5	2	Germany	1	1
Austria	1	8.5	2	Austria	0	2
Brazil	4	2.5	3	Germany	1	1
Brazil	4	2.5	3	Austria	0	2
Brazil	3	3.5	3	Germany	1	1
Brazil	3	3.5	3	Austria	0	2

In DBL2 we can use the where clause to match data (see code); in Oracle there are other matching operaters

```
select * from df, df2 where df.id = df.id;
```

In case of long names, we can use shorter aliases for table names (or use column names with aliases in the SELECT clause):

```
select * from df A, df2 B where A.id = B.id;
```

3.3 Summary

- Most databases come with built-in functions that you can use in SQL statements to perform operations on data within the database itself.
- When you work with large datasets, you may save time by using built-in functions rather than first retrieving the data into your application and then executing functions on the retrieved data.
- Use sub-queries to form more powerful queries.
- A sub-select expression helps to evaluate some built-in aggregate functions like the average function.
- Derived tables or table expressions are sub-queries where the outer query uses the results of the sub-query as a data source.

References

Grothendieck, G. 2017. "Sqldf: Manipulate r Data Frames Using SQL." https://CRAN.R-project.org/package=sqldf.

Müller, Kirill, Hadley Wickham, David A. James, and Seth Falcon. 2023. "RSQLite: SQLite Interface for r." https://CRAN.R-project.org/package=RSQLite.

Ooms, Jeroen, David James, Saikat DebRoy, Hadley Wickham, and Jeffrey Horner. 2022. "RMySQL: Database Interface and 'MySQL' Driver for r." https://CRAN.R-project.org/package=RMySQL.

R Special Interest Group on Databases (R-SIG-DB), Hadley Wickham, and Kirill Müller. 2022. "DBI: R Database Interface." https://CRAN.R-project.org/package=DBI.

Ripley, Brian, and Michael Lapsley. 2022. "RODBC: ODBC Database Access." https://CRAN.R-project.org/package=RODBC.

Urbanek, Simon. 2022. "RJDBC: Provides Access to Databases Through the JDBC Interface." https://CRAN.R-project.org/package=RJDBC.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. "Dplyr: A Grammar of Data Manipulation." https://CRAN.R-project.org/package=dplyr.