Introduction to SQL

Subqueries

Prof. Dr. Jan Kirenz

HdM Stuttgart

2019/09/01 (updated: 2019-09-03)

Setup

pw = "your_password"

```
library(DBI)
library(RPostgreSQL)

drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "postgres",
                 host = "localhost", port = 5433,
                 user = "postgres", password = pw)
```

- Note: the examples used in this presentation are based on the excellent book "A Beginner's Guide to Storytelling with Data" from Anthony DeBarros (2018).

CREATE TABLE

- This is only a part of the table (review the source code):

```
CREATE TABLE us_counties_2010 (
    geo_name varchar(90),                -- Name of the geography
    state_us_abbreviation varchar(2),    -- State/U.S. abbreviation
    summary_level varchar(3),            -- Summary Level
    region smallint,                     -- Region
    division smallint,                   -- Division
    state_fips varchar(2),               -- State FIPS code
    county_fips varchar(3),              -- County code
    area_land bigint,                    -- Area (Land) in square meters
    area_water bigint,                   -- Area (Water) in square meters
    population_count_100_percent integer,    -- Population count (100%)
    housing_unit_count_100_percent integer,  -- Housing Unit count (100%)
    internal_point_lat numeric(10,7),    -- Internal point (latitude)
    internal_point_lon numeric(10,7),    -- Internal point (longitude)

    -- This section is referred to as P1. Race:
    p0010001 integer,   -- Total population
    p0010002 integer,   -- Population of one race:
    p0010003 integer,      -- White Alone
    p0010004 integer,      -- Black or African American alone
    p0010005 integer,      -- American Indian and Alaska Native alone
    p0010006 integer,      -- Asian alone
    p0010007 integer,      -- Native Hawaiian and Other Pacific Islander alone
    p0010008 integer,      -- Some Other Race alone
```

COPY

- Data: us_counties_2010.csv

```
COPY us_counties_2010
FROM '/Users/jankirenz/Documents/HdM/Vorlesungen/DataScience/ProgrammingLanguages/SQL/sq
WITH (FORMAT CSV, HEADER);
```

```
SELECT *
FROM us_counties_2010;
```

Show 8 entries Search:

	geo_name	state_us_abbreviation	summary_level	region	division
1	Autauga County	AL	050	3	6
2	Baldwin County	AL	050	3	6
3	Barbour County	AL	050	3	6
4	Bibb County	AL	050	3	6
5	Blount County	AL	050	3	6
6	Bullock County	AL	050	3	6

Showing 1 to 6 of 6 entries Previous 1 Next

Using Subqueries

- A subquery is nested inside another query.
- It's used for a calculation or logical test that provides a value or set of data to be passed onto the main portion of the query.

Filtering with Subqueries in a WHERE clause

- A WHERE clause lets you filter query results based on criteria you provide, using an expression such as "WHERE quantity > 100"
- But this requires that you already know the value to use for comparison.
- A subquery lets you write a query that generates one or more values to use as part of an expression in a WHERE clause.

Generating Values for a Query Expression

- We want to write a query that shows which U.S. counties are at or above the 90th percentile, or top 10 percent, for population.
- Rather than writing two seperate queries (one to calculate the 90th percentile and the other to filter by counties) we can do both at once using a subquery in a WHERE clause

```
SELECT geo_name,
    state_us_abbreviation,
    p0010001
FROM us_counties_2010
WHERE p0010001 >= (SELECT percentile_cont(0.9) WITHIN GROUP
     (ORDER BY p0010001)
     FROM us_counties_2010)
ORDER BY p0010001 DESC;
```

Show 8 entries Search:

	geo_name	state_us_abbreviation	p0010001
1	Los Angeles County	CA	9818605
2	Cook County	IL	5194675
3	Harris County	TX	4092459
4	Maricopa County	AZ	3817117
5	San Diego County	CA	3095313
6	Orange County	CA	3010232
7	Kings County	NY	2504700
8	Miami-Dade County	FL	2496435

Showing 1 to 8 of 100 entries

Previous 1 2 3 4 5 … 13 Next

```
SELECT percentile_cont(0.9) WITHIN GROUP
    (ORDER BY p0010001)
    FROM us_counties_2010;
```

percentile_cont
197444.6

Using a Subquery to Identify Rows to Delete

- We can use a subquery to specify what to **remove** from a table.
- Imagine you have a table with 100 million rows that, because of its size, takes a long time to query.
- If you just want to work on a subset of the data (such as a particular state), you can make a copy of the table and delete what you don't need.
- We'll make a copy of the census table and than delete everything from that backup except the 315 counties in the top 10 percent population.

CREATE TABLE

```
CREATE TABLE us_counties_2010_top10 AS
(SELECT * FROM us_counties_2010);
```

DELETE

```
DELETE FROM us_counties_2010_top10
WHERE p0010001 < (SELECT percentile_cont(0.9) WITHIN GROUP
        (ORDER BY p0010001)
        FROM us_counties_2010_top10);
```

```
SELECT COUNT(*)
FROM us_counties_2010_top10;
```

count
315

```
SELECT COUNT(*)
FROM us_counties_2010;
```

count
3143

Subquery as a derived table in a FROM clause

- In this query we want to find the **average** and **median** population of U.S. counties as well as the **difference** between them.
- We need to calculate the average and the median, and then we need to subtract the two.
- We use the AVG and PERCENTILE functions to find th average and median of the census table's p0010001 total polulation column and name each column with an alias (AS).
- Then we name the subquery calcs (AS calcs), so we can reference it as a table in the main query.
- Subtracting the median from the average (both are returned by the subquery), is done in the main query (first SELECT clause).
- Then the main query rounds the result and labels it as median_average_diff.

Median & average

```
SELECT ROUND(calcs.average, 0) as average,
       calcs.median,
       ROUND(calcs.average - calcs.median, 0) AS median_average_diff
FROM (SELECT AVG(p0010001) AS average,
        percentile_cont(.5)
        WITHIN GROUP (ORDER BY p0010001)::numeric(10,1) AS median
      FROM us_counties_2010)
      AS calcs;
```

average	median	median_average_diff
98233	25857	72376

Median & average

- The difference between median and average is nearly three times the size of the median.
- That shows that a relatively small number of high-population counties push the average county size over 98000.

Adding a subquery to column list

- You can generate new columns of data with subqueries by placing a subquery in the column list after SELECT.
- For example: the query below selects geo_name and total population column p0010001 from us_counties_2010, and then adds a subquery to add the median of all counties to each row in the new column us_median:

```
SELECT geo_name,
    state_us_abbreviation AS st,
    p0010001 AS total_pop,
    (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY p0010001)
     FROM us_counties_2010) AS us_median
FROM us_counties_2010;
```

geo_name	st	total_pop	us_median
Autauga County	AL	54571	25857
Baldwin County	AL	182265	25857
Barbour County	AL	27457	25857
Bibb County	AL	22915	25857
Blount County	AL	57322	25857
Bullock County	AL	10914	25857

Using a subquery expression in a calculation

- It can also be useful to generate values that indicate how much each country's population deviates from the median value:

```
SELECT geo_name,
      state_us_abbreviation AS st,
      p0010001 AS total_pop,
      (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY p0010001)
       FROM us_counties_2010) AS us_median,
      p0010001 - (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY p0010001)
                  FROM us_counties_2010) AS diff_from_median
FROM us_counties_2010
WHERE (p0010001 - (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY p0010001)
                   FROM us_counties_2010))
      BETWEEN -1000 AND 1000;
```

Using a subquery expression in a calculation

- The added subquery subtracts the first subquery's result (us_median) from p0010001, the total population and creates a new column called diff_from_median.
- By repeating the subquery expression in the WHERE clause and filter results using BETWEEN, we narrow the results further to show only counties which population falls within 1000 of the median (counties close to the median population)

Show 8 entries Search:

	geo_name	st	total_pop	us_median	diff_from_median
1	Cherokee County	AL	25989	25857	132
2	Clarke County	AL	25833	25857	-24
3	Geneva County	AL	26790	25857	933
4	Cleburne County	AR	25970	25857	113
5	Johnson County	AR	25540	25857	-317
6	Ouachita County	AR	26120	25857	263
7	Montezuma County	CO	25535	25857	-322
8	Washington County	FL	24896	25857	-961

Showing 1 to 8 of 71 entries

Previous 1 2 3 4 5 … 9 Next

Subquery Expressions

- We can use subqueries to **filter rows** by evaluating whether a **condition** evaluates as TRUE or FALSE.

Creating tables

```
CREATE TABLE retirees (
    id int,
    first_name varchar(50),
    last_name varchar(50)
);
```

```
INSERT INTO retirees
VALUES (2, 'Lee', 'Smith'),
       (4, 'Janet', 'King');
```

Creating tables

```
CREATE TABLE employees (
    emp_id bigserial,
    first_name varchar(100),
    last_name varchar(100),
    salary integer,
    dept_id integer REFERENCES departments (dept_id),
    CONSTRAINT emp_key PRIMARY KEY (emp_id),
    CONSTRAINT emp_dept_unique UNIQUE (emp_id, dept_id)
);
```

```
INSERT INTO employees (first_name, last_name, salary, dept_id)
VALUES
    ('Nancy', 'Jones', 62500, 1),
    ('Lee', 'Smith', 59300, 1),
    ('Soo', 'Nguyen', 83000, 2),
    ('Janet', 'King', 95000, 2);
```

```
SELECT *
FROM employees;
```

emp_id	first_name	last_name	salary	dept_id
1	Nancy	Jones	62500	1
2	Lee	Smith	59300	1
3	Soo	Nguyen	83000	2
4	Janet	King	95000	2

```
SELECT *
FROM retirees;
```

id	first_name	last_name
2	Lee	Smith
4	Janet	King

Generating Values for the IN Operator

- The following query uses a subquery to generate id values from a retirees table, and then uses that list for the IN operator in the WHERE clause.
- The NOT IN expression does the opposite to find employees whose id value does not appear in retirees.

```
SELECT first_name, last_name
FROM employees
WHERE emp_id IN (
      SELECT id
      FROM retirees);
```

first_name	last_name
Lee	Smith
Janet	King

Generating Values for the IN Operator

- The output shows the names of employees who have id values that match those in retirees.
- Note: The presence of NULL values in a subquery result set will cause a query with a NOT IN expression to return no rows. If your data contains NULL values, use the WHERE NOT EXISTS expression described in the next query.

Checking Whether Values Exist

- EXIST is a TRUE/FALSE test
- it returns TRUE if the subquery in parentheses returns at least one row
- if it returns no rows, EXISTS evaluates to FALSE

```
SELECT first_name, last_name
FROM employees
WHERE EXISTS (
   SELECT id
   FROM retirees);
```

first_name	last_name
Nancy	Jones
Lee	Smith
Soo	Nguyen
Janet	King

- To mimic the behavior of IN and limit names to where the subquery after EXISTS finds at least one corresponding id value in retirees.
- Using a correlated subquery to find matching values from employees in retirees:

```
SELECT first_name, last_name
FROM employees
WHERE EXISTS (
   SELECT id
   FROM retirees
   WHERE id = employees.emp_id);
```

first_name	last_name
Lee	Smith
Janet	King

- This approach is helpful if you need to join on more than one column, which you can't do with the IN expression.
- We can also use NOT with EXISTS, for example to find employees with no corresponding record in retirees:

```
SELECT first_name, last_name
FROM employees
WHERE NOT EXISTS (
   SELECT id
   FROM retirees
   WHERE id = employees.emp_id);
```

- This technique is helpful for assessing whether a data set is complete.

first_name	last_name
Nancy	Jones
Soo	Nguyen

Common Table Expressions

- A "CTE" (Common Table Expression) is temporary result set that you can reference within another SELECT, INSERT, UPDATE, or DELETE statement.
- A CTE always returns a result set.
- They are used to simplify queries.
- Informally called WITH clause.
- Using a CTE, you can define one or more tables up front with subqueries. Then you can query the table results as often as needed in a main query that follows.

Common Table Expressions

- Using a simple CTE to find large counties:

```
WITH
   large_counties (geo_name, st, p0010001)
AS
   (
      SELECT geo_name, state_us_abbreviation, p0010001
      FROM us_counties_2010
      WHERE p0010001 >= 100000
   )
SELECT st, count(*)
FROM large_counties
GROUP BY st
ORDER BY count(*) DESC;
```

Common Table Expressions

- The WITH ... AS block defines the CTE's temporary table large_counties
- After WITH, we name the table and list its column names in parentheses
- Unlike column definitions in a CREATE TABLE statement, we don't need to provide data types, because the temporary table inherits those from the subquery, which is enclosed in parentheses after AS.
- The subquery must return the same number of columns as defined in the temporary table, but the column names don't need to match.
- The column list is optional if you're not renaming columns, although it is still good for clarity even if you don't rename columns.
- The main query (last SELECT statement) counts and groups the rows in large_counties by st, and then orders by the count in descending order.

st	count
TX	39
CA	35
FL	33
PA	31
OH	28
NY	28

- You can get the same result using a SELECT query instead of a CTE:

```
SELECT state_us_abbreviation, COUNT(*)
FROM us_counties_2010
WHERE p0010001 >= 100000
GROUP BY state_us_abbreviation
ORDER BY COUNT(*) DESC;
```

state_us_abbreviation	count
TX	39
CA	35
FL	33
PA	31
NY	28
OH	28

Why use CTE?

- By using a CTE, you can pre-stage subsets of data to feed into larger query for more complex analysis.
- Also, you can reuse each table defined in a CTE in multiple places in the main query, which means you don't have to repeat the SELECT query each time.
- Sometimes the code is more readable

Cross Tabulations

- Cross Tabulations provide a simple way to summerize and compare variables by displaying them in a table layout, or matrix.
- In a matrix, rows represent another variable, columns represent another variable, and each cell where a row and column intersects holds a value, such as a count or percentage.
- Cross Tabulations are also called **pivot tables** or **crosstabs**.
- They are often used to report summeries of survey results or to compare sets of variables.

Installing the crosstab Function

```
CREATE EXTENSION tablefunc;
```

- Creating and filling the ice_cream_survey table

```
CREATE TABLE ice_cream_survey (
    response_id integer PRIMARY KEY,
    office varchar(20),
    flavor varchar(20)
);
```

- Data: ice_cream_survey.csv

```
COPY ice_cream_survey
FROM '/Users/jankirenz/Documents/HdM/Vorlesungen/DataScience/ProgrammingLanguages/SQL/sq
WITH (FORMAT CSV, HEADER);
```

- Inspecting the data

```
SELECT *
FROM ice_cream_survey
LIMIT 5;
```

response_id	office	flavor
1	Uptown	Chocolate
2	Midtown	Chocolate
3	Downtown	Strawberry
4	Uptown	Chocolate
5	Midtown	Chocolate

Generating the ice cream survey tab

```
SELECT *
FROM crosstab('SELECT office,
                    flavor,
                    count(*)
               FROM ice_cream_survey
               GROUP BY office, flavor
               ORDER BY office',

              'SELECT flavor
               FROM ice_cream_survey
               GROUP BY flavor
               ORDER BY flavor')

AS (office varchar(20),
    chocolate bigint,
    strawberry bigint,
    vanilla bigint);
```

Generating the ice cream survey tab

- The first statement selects everything from the contents of the crosstab function.
- We placed two subqueries inside the crosstab function.
- The first subquery generates the data for the crosstab and has three required columns.
- The first column, office, supplies the row names for the crosstab.
- The second column, flavor, supplies the category columns.
- The third column supplies the values for each cell where row and column intersect in the table, here we want the intersecting cells to show a COUNT(*) of each flavor selected at each office
- This first subquery creates a simple aggregated list (below)

office	chocolate	strawberry	vanilla
Downtown	23	32	19
Midtown	41	NA	23
Uptown	22	17	23

Generating the ice cream survey tab

Step by step

```
SELECT office,
       flavor,
       count(*)
  FROM ice_cream_survey
 GROUP BY office, flavor
 ORDER BY office
```

office	flavor	count
Downtown	Chocolate	23
Downtown	Vanilla	19
Downtown	Strawberry	32
Midtown	Vanilla	23
Midtown	Chocolate	41
Uptown	Strawberry	17

Generating the ice cream survey tab

Step by step

```
SELECT flavor
        FROM ice_cream_survey
        GROUP BY flavor
        ORDER BY flavor;
```

- The second subquery produces the set category names for the columns.
- The crosstab function requires that the second subquery return only one column, so here we use SELECT to retrieve the flavor column, and we use GROUP BY to return that column's unique values

flavor
Chocolate
Strawberry
Vanilla

Generating the ice cream survey tab

Step by step

- Then we specify the names and data types of the crosstab's output columns following the AS keyword
- The list must match the row and column names in the order the subqueries generate them

Tabulating City Temperature Readings

Creating and filling a temperature_readings table

```
CREATE TABLE temperature_readings (
    reading_id bigserial PRIMARY KEY,
    station_name varchar(50),
    observation_date date,
    max_temp integer,
    min_temp integer
);
```

Tabulating City Temperature Readings

- Data: temperature_readings.csv

```
COPY temperature_readings
    (station_name, observation_date, max_temp, min_temp)
FROM '/Users/jankirenz/Documents/HdM/Vorlesungen/DataScience/ProgrammingLanguages/SQL/sq
WITH (FORMAT CSV, HEADER);
```

Inspecting Data

```
SELECT *
FROM temperature_readings
LIMIT 5;
```

reading_id	station_name	observation_date	max_temp	min_temp
1	CHICAGO NORTHERLY ISLAND IL US	2016-01-01	31	20
2	CHICAGO NORTHERLY ISLAND IL US	2016-01-02	34	23
3	CHICAGO NORTHERLY ISLAND IL US	2016-01-03	32	26
4	CHICAGO NORTHERLY ISLAND IL US	2016-01-04	32	27

Inspecting Data

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

count
1077

- Generating the temperature readings crosstab

```
SELECT *
FROM crosstab('SELECT
                station_name,
                date_part("month", observation_date),
                percentile_cont(.5)
                   WITHIN GROUP (ORDER BY max_temp)
              FROM temperature_readings
              GROUP BY station_name,
                     date_part("month", observation_date)
              ORDER BY station_name',

             'SELECT month
              FROM generate_series(1,12) month')

AS (station varchar(50),
    jan numeric(3,0),
    feb numeric(3,0),
    mar numeric(3,0),
    apr numeric(3,0),
    may numeric(3,0),
    jun numeric(3,0),
    jul numeric(3,0),
    aug numeric(3,0),
    sep numeric(3,0),
    oct numeric(3,0),
    nov numeric(3,0),
```

station	jan	feb	mar	apr	may	jun	jul	aug	sep	oct	nov	dec
CHICAGO NORTHERLY ISLAND IL US	34	36	46	50	66	77	81	80	77	65	57	35
SEATTLE BOEING FIELD WA US	50	54	56	64	66	71	76	77	69	62	55	42
WAIKIKI 717.2 HI US	83	84	84	86	87	87	88	87	87	86	84	82

Generating the temperature readings crosstab

Step by step

- First subquery: generates the data for the crosstab, calculating the median maximum temperature for each month.
- It supplies the three required columns.
- The first column, station_name, names the rows.
- The second column uses the date_part function to extract the month from observation _date, which provides the crosstab columns.
- Then we use percentile_cont(.5) to find the 50th percentile, or the median, of the max_temp.
- We group by station and month so we have a median max_temp for each month at each station.

Generating the temperature readings crosstab

Step by step

```
SELECT
        station_name,
        date_part('month', observation_date),
        percentile_cont(.5)
          WITHIN GROUP (ORDER BY max_temp)
    FROM temperature_readings
    GROUP BY station_name,
          date_part('month', observation_date)
    ORDER BY station_name;
```

station_name	date_part	percentile_cont
CHICAGO NORTHERLY ISLAND IL US	1	34
CHICAGO NORTHERLY ISLAND IL US	2	36
CHICAGO NORTHERLY ISLAND IL US	3	46
CHICAGO NORTHERLY ISLAND IL US	4	50

Generating the temperature readings crosstab

Step by step

- Second subquery: produces the set of category names for the columns.
- The generate_series function creates a list of numbers from 1 to 12 that match the month numbers date_part extracts from observation_date:

```
SELECT month
FROM generate_series(1,12) month;
```

month
1
2
3
4
5
6
7
8
9
10
11
12

- Following AS, we provide the names and data types for the crosstab's output columns:

```
SELECT *
FROM crosstab('SELECT
                station_name,
                date_part("month", observation_date),
                percentile_cont(.5)
                   WITHIN GROUP (ORDER BY max_temp)
              FROM temperature_readings
              GROUP BY station_name,
                     date_part("month", observation_date)
              ORDER BY station_name',

             'SELECT month
              FROM generate_series(1,12) month')
AS (station varchar(50),
    jan numeric(3,0),
    feb numeric(3,0),
    mar numeric(3,0),
    apr numeric(3,0),
    may numeric(3,0),
    jun numeric(3,0),
    jul numeric(3,0),
    aug numeric(3,0),
    sep numeric(3,0),
    oct numeric(3,0),
    nov numeric(3,0),
```

station	jan	feb	mar	apr	may	jun	jul	aug	sep	oct	nov	dec
CHICAGO NORTHERLY ISLAND IL US	34	36	46	50	66	77	81	80	77	65	57	35
SEATTLE BOEING FIELD WA US	50	54	56	64	66	71	76	77	69	62	55	42
WAIKIKI 717.2 HI US	83	84	84	86	87	87	88	87	87	86	84	82

- The table shows the median maximum temperature each month for each station

Reclassifying Values with CASE

- CASE is a conditional expression, meaning it let's you add some "if this, then ..." logic to your query.
- The CASE syntax follows this pattern:

```
CASE WHEN condition THEN result
     WHEN another_condition THEN result
     ELSE result
END
```

CASE

- Conditions can be any expression that the database can evaluate as TRUE or FALSE.
- If the condition is TRUE, the case statement returns the result and stops checking any further conditions.
- If the condition is FALSE, the database moves on to evaluate the next condition.
- We can also provide an optional **ELSE** statement to return a result in case no condition evaluates as TRUE.
- Without an **ELSE** clause, the statement would return a **NULL** when no conditions are TRUE

Reclassifying temperature data with CASE

- We are creating five ranges for the max_temp column in temperature_reading, which we define using comparison operators.
- The CASE statement evaluates each value to find whether any of the five expressions are TRUE.
- If TRUE, the statement outputs the appropriate text.
- If none of the statements is TRUE, the ELSE clause assigns the value to the category Inhumane.

Reclassifying temperature data with CASE

```
SELECT max_temp,
      CASE WHEN max_temp >= 90 THEN 'Hot'
         WHEN max_temp BETWEEN 70 AND 89 THEN 'Warm'
         WHEN max_temp BETWEEN 50 AND 69 THEN 'Pleasant'
         WHEN max_temp BETWEEN 33 AND 49 THEN 'Cold'
         WHEN max_temp BETWEEN 20 AND 32 THEN 'Freezing'
         ELSE 'Inhumane'
       END AS temperature_group
FROM temperature_readings;
```

max_temp	temperature_group
31	Freezing
34	Cold
32	Freezing
32	Freezing
34	Cold
38	Cold

Using CASE in a Common Table Expression (CTE)

- The query before is a good example of a preprocessing step you would use in a CTE.
- Now that we've grouped the temperatures in categories, let's count the groups by city in a CTE to see how many days of the year fall into each temperature category.

Using CASE in a CTE

- This code reclassifies the temperatures, and then counts and groups by station name to find general climate classifications of each city.
- WITH defines the CTE of temps_collapsed, which has two columns: station_name and max_temperature_group
- We then run a SELECT query on the CTE, performing straightforward COUNT(*) and GROUP BY operations on both columns.

```
WITH temps_collapsed (station_name, max_temperature_group) AS
   (SELECT station_name,
         CASE WHEN max_temp >= 90 THEN 'Hot'
            WHEN max_temp BETWEEN 70 AND 89 THEN 'Warm'
            WHEN max_temp BETWEEN 50 AND 69 THEN 'Pleasant'
            WHEN max_temp BETWEEN 33 AND 49 THEN 'Cold'
            WHEN max_temp BETWEEN 20 AND 32 THEN 'Freezing'
            ELSE 'Inhumane'
          END
   FROM temperature_readings)

SELECT station_name, max_temperature_group, count(*)
FROM temps_collapsed
GROUP BY station_name, max_temperature_group
ORDER BY station_name, count(*) DESC;
```

station_name	max_temperature_group	count
CHICAGO NORTHERLY ISLAND IL US	Warm	133
CHICAGO NORTHERLY ISLAND IL US	Cold	92
CHICAGO NORTHERLY ISLAND IL US	Pleasant	91
CHICAGO NORTHERLY ISLAND IL US	Freezing	30
CHICAGO NORTHERLY ISLAND IL US	Inhumane	8
CHICAGO NORTHERLY ISLAND IL US	Hot	8
SEATTLE BOEING FIELD WA US	Pleasant	198
SEATTLE BOEING FIELD WA US	Warm	98
SEATTLE BOEING FIELD WA US	Cold	50
SEATTLE BOEING FIELD WA US	Hot	3
WAIKIKI 717.2 HI US	Warm	361
WAIKIKI 717.2 HI US	Hot	5

Using CASE in a CTE / Step by step

```
SELECT station_name,
        CASE WHEN max_temp >= 90 THEN 'Hot'
            WHEN max_temp BETWEEN 70 AND 89 THEN 'Warm'
            WHEN max_temp BETWEEN 50 AND 69 THEN 'Pleasant'
            WHEN max_temp BETWEEN 33 AND 49 THEN 'Cold'
            WHEN max_temp BETWEEN 20 AND 32 THEN 'Freezing'
            ELSE 'Inhumane'
         END
   FROM temperature_readings
```

station_name	case
CHICAGO NORTHERLY ISLAND IL US	Freezing
CHICAGO NORTHERLY ISLAND IL US	Cold
CHICAGO NORTHERLY ISLAND IL US	Freezing
CHICAGO NORTHERLY ISLAND IL US	Freezing
CHICAGO NORTHERLY ISLAND IL US	Cold
CHICAGO NORTHERLY ISLAND IL US	Cold
CHICAGO NORTHERLY ISLAND IL US	Cold

Using CASE in a CTE / Step by step

```
WITH temps_collapsed (station_name, max_temperature_group) AS
   (SELECT station_name,
        CASE WHEN max_temp >= 90 THEN 'Hot'
           WHEN max_temp BETWEEN 70 AND 89 THEN 'Warm'
           WHEN max_temp BETWEEN 50 AND 69 THEN 'Pleasant'
           WHEN max_temp BETWEEN 33 AND 49 THEN 'Cold'
           WHEN max_temp BETWEEN 20 AND 32 THEN 'Freezing'
           ELSE 'Inhumane'
         END
   FROM temperature_readings)

SELECT station_name, max_temperature_group, count(*)
FROM temps_collapsed
GROUP BY station_name, max_temperature_group
ORDER BY station_name, count(*) DESC;
```

station_name	max_temperature_group	count
CHICAGO NORTHERLY ISLAND IL US	Warm	133
CHICAGO NORTHERLY ISLAND IL US	Cold	92
CHICAGO NORTHERLY ISLAND IL US	Pleasant	91

Thank you!

Prof. Dr. Jan Kirenz

HdM Stuttgart
Nobelstraße 10
70569 Stuttgart