Chapter 2

Basic Manipulations

Contents

Up to this point we have refrained from transforming any of the data that is being returned in our queries. In this module we being working on manipulating the data that is being returned via functions, renaming and other methods. Importantly, none of what we are doing changes the underlying data; it simply transforms what is being returned to the client.

Before manipulating, however, we need to understand data within a relational database and how it is represented. In particular, we need to understand "types".

1 Types

- Relational databases are "strongly" typed, meaning that there are strict rules around what operations can be performed on what data.
- As in other computer languages, types determines both what operations are available and how operations behave as a function of the data contained therein.
- In Relational Databases, columns are typed and set when a table is created. A column can only be a single type.
- Relational databases support a variety of different data types. In this section we will discuss the most commonly used ones, a hierarchy of which can be found in Figure 2.1.

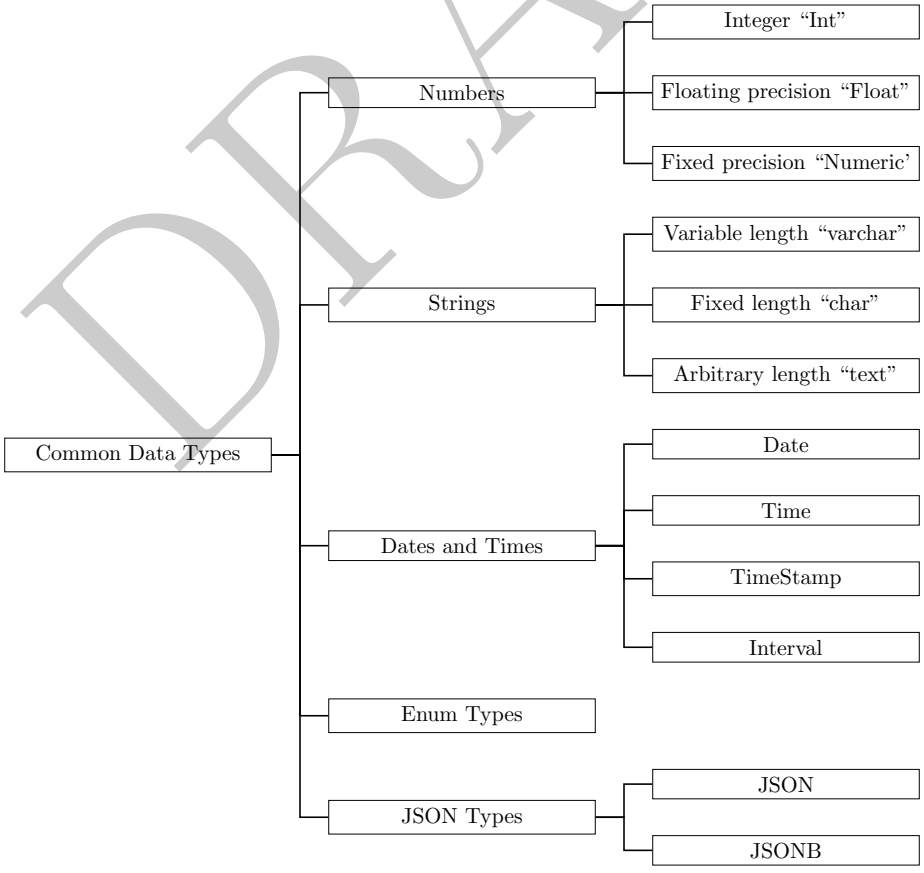

Figure 2.1: Common relational database data types

Numbers

There are three "styles" of numbers:

1. **Integer:** These are whole numbers and there are actually 3 different types: smallint (2 bytes, can store -32,768 to +32,767 (2^{15})), int (4 bytes, can store -2,147,483,648 to +2,147,483,647 (2^{31})) and bigint (8 bytes, can store -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (2^{63})).
2. **Float:** A floating point number is an inexact, variable precision numeric type, usually coming in two flavors: real (4 bytes, 1E-37 to 1E+37 with a precision of at least 6 decimal digits) and double (8 bytes, 1E-307 to 1E+308 with a precision of at least 15 digit).
3. **Numeric:** A numeric has a user-defined fixed precision (like 2 decimal places). They vary in size and type depending on the amount of precision required. An example use of fixed precision is storing information about money; there is a fixed cut-off (penny) of precision.

In practice database administrators tend to stick to using integers and floats with an occasional numeric types.

Strings

The three most common string types used are:

1. **Variable length:** The "varchar" type is used for variable length strings, but with a maximum number of specified characters. For example, a varchar(10) can contain any string, as long as the number of characters is less than or equal to 10.
2. **Fixed length:** The "char" type is used for fixed length strings. For example, a char(10) can contain any string, as long as the number of characters is less than or equal to 10. The difference between char and varchar is that this type always reserves space for additional characters, up to the the max, while a varchar does not. So, to store the names "Nick", "John" and "Reggie" as a varchar(6) would take (approximately) $(4 + 4 + 6 = 16$ bytes) while storing those same names as a char(6) would take (approximately) $6 + 6 + 6 = 18$ bytes.
3. **Arbitrary length:** The "text" type (sometimes called blob) is used for strings of arbitrary length. For example, if you wanted to store yelp comments you would use a text field, since the comments can be any length. Text fields are generally avoided when another type can be used due to storage efficiency.

Enum

- For categorical data databases use what is called an "enum" or "enumerated" type.
- This type stores the data as an integer which also has a "map" that maps those numbers to specific values.
- The classic version of this is gender. Consider a survey with the following options:

	No. Chars	Enum Value
Woman	5	1
Man	3	2
Transgender	11	3
Non-binary/non-conforming	25	4
Prefer not to respond	21	5

- In this example, storing the data as an enum would save a ton of space over storing it as text.

- The downside is increased complexity and issues with comparisons (do you compare based on the map or on the text value)?
- All modern databases have a version of this, but we won't get too much into the details in this course.

JSON

- Modern databases usually have two different options for storing JSON information: a raw representation and a binary representation.
- The raw representation is just a text blob that, by calling it "JSON" you get access to special functions only available to JSON objects, specifically functions around keys and values.
- The JSONB representation is a further parsed, binary representation of the JSON data. JSONB data (usually) is slower to load into the database due to the additional type conversions, but faster to do lookup operations on.
- All modern databases support JSON path (sometimes called JSONpath) syntax for accessing operators. This will be discussed later.

Dates

We will hold off on discussing dates until Module 6.

2 Renaming a Column

- The first thing we will learn to do is change the name of tables and columns that are being returned.
- We sometimes want to rename columns. To do this, we use the `AS` operator:

```
select
    registrations as reg2
from
    cls.cars;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

  The query above will return a single column, with the name `reg2`.

- We can also use it to rename tables, though this won't be useful for a few weeks!

```
select
    registrations as reg2
from
    cls.cars as c2;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

- We can actually just skip the `AS` completely, thought it isn't recommended since it can make the query more difficult to read.

```
select
    registrations  reg2
from
    cls.cars  c2;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

3 Basic Mathematical Manipulations, ABS and LEAST/GREATEST

- If, instead of selecting a column directly from the table, we put down a single value, then that value will be repeated for each row returned.
- For example, consider the following query:

```
select
    1 as v1, 2 as v2, 'Nick' as name, vehicletype
from
    cls.cars;


  v1    v2  name    vehicletype
----  ----  ------  --------------
   1     2  Nick    Bus
   1     2  Nick    Moped
   1     2  Nick    Truck
   1     2  Nick    Travel Trailer
   1     2  Nick    Truck
[...]
```

- Note that the data is repeated once for each row and no rows are being generated.
- We can also manipulate the data that is being returned on a row-by-row basis by using functions within the select.
- For example, we can do basic math functions:

```
select
    registrations + 10 as reg2
    , registrations
from
    cls.cars


  reg2    registrations
------  ---------------
    15                5
   208              198
  5030             5020
   376              366
  2517             2507
[...]
```

Note that what this does is create a synthetic column of the number ten (repeated for each row) and then adds that to the column "reg". The result is that each entry in "reg2" is equal to "reg" plus 10.

- All standard mathematical operations $(+, -, /, *)$ are all supported and math can be done between columns, such as:

```
select
    registrations + 10 as reg2
    , annualfee * annualfee as annualfee_sq
    , registrations
from
    cls.cars;


  reg2      annualfee_sq    registrations
------  ----------------  ---------------
    15  462400                          5
   208       1.921e+06                198
  5030       9.60083e+10             5020
   376       3.39039e+08              366
  2517       1.7873e+10              2507
[...]
```

- What if we fail to rename the column with `AS`? The database will generate a column name for us:

```
select
    registrations + 10
    , annualfee * annualfee
    , registrations
from
    cls.cars;


  ?column?          ?column?    registrations
----------  ----------------  ---------------
        15  462400                          5
       208       1.921e+06                198
      5030       9.60083e+10             5020
       376       3.39039e+08              366
      2517       1.7873e+10              2507
[...]
```

  In this case the database has no idea what to name the column so calls it `?column?`.

- SQL also has more advanced functions, many of which are similar to Excel. For example, the absolute value function (`ABS`), which returns the magnitude of a number without regard for its sign, can be used to return a modified column:

```
SELECT
    abs( registrations - 1000 ) as abs_reg, registrations
FROM
    cls.cars;


  abs_reg    registrations
---------  ---------------
      995                5
      802              198
     4020             5020
      634              366
     1507             2507
[...]
```

returns two columns from cars. The first is the absolute value of 1,000 subtracted from registrations and the second is the registrations number.

- As with the other SQL functions we have seen, these can be used within a `WHERE` clause:

```
SELECT
    *
FROM
    cls.cars
WHERE
    abs(registrations - 1000) <= 20;


  year  countyname    motorvehicle    vehiclecat    vehicletype      tonnage      [...]
------  ------------  --------------  ------------  ---------------  ---------  - [...]
  2005  Hardin        Yes             Motorcycle    Motorcycle                    [...]
  2013  Mahaska       No              Trailer       Regular Trailer               [...]
  2008  Boone         No              Trailer       Travel Trailer                [...]
  2016  Marion        No              Trailer       Semi Trailer                  [...]
  2012  Webster       No              Trailer       Semi Trailer                  [...]
[...]
```

which returns the 231 rows where the number of registrations are between 980 and 1,020.

- The functions `LEAST` and `GREATEST` do exactly what they say – they return the highest and lowest value in a particular set of observations. Note that `LEAST` and `GREATEST` only work within a single row:

```
select
    countyname
    , abs(registrations - 100) as c1
    , abs(registrations - 30) as c2
    , registrations
    , least( abs(registrations - 100), abs(registrations - 30)) as calc_1
    , greatest( abs(registrations - 100), abs(registrations - 30)) as calc_2
from cls.cars
where registrations >= 64 and registrations <= 66
and countyname = 'Wright';


countyname      c1    c2    registrations    calc_1    calc_2
------------  ----  ----  ---------------  --------  --------
Wright          36    34               64        34        36
Wright          34    36               66        34        36
Wright          35    35               65        35        35
Wright          36    34               64        34        36
Wright          36    34               64        34        36
[...]
```

4 Queries without a FROM Clause and Singletons

- SQL allows for queries without a `FROM`. When doing this, no columns can be referenced, but the query will be executed as a single expression. This is handy when running tests, such as if we didn't understand the `ABS` function:

```
select abs( -5 ) as calc;


  calc
------
     5
```

We can do this with almost any SQL function, including mathematical operations:

```
select 5 * 10 as calc;


  calc
------
    50
```

- If a query returns a single value, which we will call a *singleton* in this class, then we can treat that value as what it returns. The code below, for example, returns twice the largest registrations:

```
 select 2 * (select registrations from cls.cars order by 1 desc limit 1) as calc;


  calc
------
437950
```

- We can also use this in a `WHERE` clause:

```
select registrations
from
    cls.cars
where
    registrations >
        10*(select registrations from cls.cars order by 1 asc limit 1)
order by registrations asc
limit 10;


  registrations
---------------
             11
             11
             11
             11
             11
[...]
```

This query will return the registrations in `cls.cars` which are 10 times more than the smallest value. It will only return the smallest 10 of those rows.

5 String Functions: LEFT, RIGHT, LOWER, UPPER, LENGTH, TRIM and CONCAT

- The string operators `LEFT` and `RIGHT` behave just as in Excel: they take the left or right characters of a string. For example:

```
select left( 'THIS STRING', 4) as left_4;


left_4
--------
THIS
```

will return 'THIS' since it is the four left most letters of the string in question.

- Both the `LEFT` and `RIGHT` commands take the same inputs: a string and the number of characters to cut:

```
select
    countyname
    , left( countyname, 4) as left_4
    , right( countyname, 4) as right_4
from
    cls.cars;


countyname    left_4    right_4
------------  --------  ---------
Ida           Ida       Ida
Jasper        Jasp      sper
Harrison      Harr      ison
Palo Alto     Palo      Alto
Adair         Adai      dair
[...]
```

- Two other string functions that behave similarly to Excel are `LOWER` and `UPPER`, which return a lowercase and uppercase version of a string column:

```
select
    countyname
    , lower(countyname) as lc
    , upper(countyname) as uc
from
    cls.cars;


countyname    lc         uc
------------  ---------  ---------
Ida           ida        IDA
Jasper        jasper     JASPER
Harrison      harrison   HARRISON
Palo Alto     palo alto  PALO ALTO
Adair         adair      ADAIR
[...]
```

  will return the countyname (with capital casing, such as "Adair") and also a lower- and upper-case version of the countyname.

- Note that we can nest functions:

```
select
    lower( left(countyname, 4)) as ll4
    ,countyname
from
    cls.cars;


ll4    countyname
-----  ------------
ida    Ida
jasp   Jasper
harr   Harrison
palo   Palo Alto
adai   Adair
[...]
```

- the LENGTH command returns the length of a string.[1] For example:

```
select
    countyname, length(countyname) as len
from
    cls.cars;


countyname      len
------------  -----
Ida               3
Jasper            6
Harrison          8
Palo Alto         9
Adair             5
[...]
```

- The `TRIM` command can be used to remove letters from a string. The default behavior is to remove spaces, but it is possible to use it for other things.

```
 select '     aaaa' as val1, trim('     aaa     ') as trm;


val1    trm
------  -----
aaaa    aaa
```

  Importantly the leading and trailing spaces have been removed from the string. Note that the commands `LTRIM` and `RTRIM` do what they are expected to do – trim from only a single side.

- To put two strings together, similar to an "&" in Excel, we can use a concatenation operator, "||". For example, the following query will return a single column with the countyname twice.

[1] In MS-SQL this is LEN, not LENGTH.

```
select
    countyname || countyname as str_calc
from
    cls.cars;


str_calc
------------------
IdaIda
JasperJasper
HarrisonHarrison
Palo AltoPalo Alto
AdairAdair
[...]
```

- We can also put a constant into the string concatenation to modify it, as in the following example:

```
select
    'County Name = ' || countyname as str_calc
from
    cls.cars;


str_calc
-----------------------
County Name = Ida
County Name = Jasper
County Name = Harrison
County Name = Palo Alto
County Name = Adair
[...]
```

Different variants of SQL use different operators for string concatenation:

SQL Variant	Syntax	Example
MySQL	concat()	select concat(col1, col2) from tablename;
MS-SQL	+	select col1 + col2 from tablename;

- A final useful command for parsing strings is `LENGTH`, which returns the length of a string. We can use this with right and left to uppercase the last letter of a string:

```
select
    left(countyname, length(countyname) - 1)
        || upper( right( countyname, 1) ) as lastUpper
from
    cls.cars;


lastupper
-----------
IdA
JaspeR
HarrisoN
Palo AltO
AdaiR
[...]
```

which returns a list of countynames with both the first and last letter upper-cased!

6 ROUND and Changing Types (CAST)

7 CAST and changing types

- It can be the case that you want to switch data types and then do operations on them.
- To do this we use the "CAST" operator, which takes a column and a target data type as its inputs. Unlike other functions, however, the word "as" is used to split the inputs. Consider the following examples:

```
select 125.5 + 4 as ans;


  ans
-----
129.5
```

```
select '125.5' + 4 as ans;
ERROR:  invalid input syntax for integer: "125.5"
LINE 1: select '125.5' + 4 as ans;
^
```

```
select cast( '125.5' as float) + 4 as ans;


  ans
-----
129.5
```

The first query returns the expected answer while the second errors out because it tries to add a string and an integer. The third query uses the cast operator to change the data type.

- Rather than using `CAST` PostgreSQL provides a double colon operator to do the same thing:

```
select '123.5'::float + 4 as ans;


  ans
-----
127.5
```

- Finally, keep in mind that PostgreSQL will attempt to do many conversions, even if you don't explicitly specify them. For example:

```
select '123' + 4 as ans;


  ans
-----
  127
```

  Surprisingly, the database is able to make this conversion and thus does the math correctly.

- One commonly used function is the `ROUND` command which rounds a number.
- Lets say that we wanted to get the `annualfee` and `registrations` rounded to the nearest 100. In this case we could start by doing the following:

```
select
    round( registrations, -2 ) as rounded_reg
    , registrations
from
    cls.cars;


  rounded_reg    registrations
-------------  ---------------
            0                5
          200              198
         5000             5020
          400              366
         2500             2507
[...]
```

  As from the results above, the `ROUND` commands rounds numbers to the place specified in the integer following the value to be rounded. In this example the rounding occurs to the -2 position which is the hundreds place.

  Moving to `annualfee`, we could write it as:

```
select
    round( annualfee, -2 ) as rounded_af
    , annualfee
from
    cls.cars;
```

which would return:

```
ERROR:  function round(double precision, integer) does not exist
LINE 4:     , round( annualfee, -2 ) as rounded_af
              ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
```

Why does this return an error?!?!

The round command is *type* dependent. If you have an integer or a numeric type, the syntax is `ROUND( column, integer )` where the integer determines where to round the value. If the integer is positive, it will round to values *after* the decimal while negative integers in the `ROUND` will return values rounded to places before the decimal, as in the example above. On the other hand, floats (called "double precision" in the error) do not accept a second argument and will only round to the nearest integer!

- So how do we handle rounding to the nearest hundreds for a float? There are two options: we either transform the column to use the `ROUND` command on floats or we `CAST` the float as a different type (either numeric or integer) and then use the available parameters therein.
- The first option:

```
 select
     round(registrations, -2) as rounded_reg
    , registrations
    , 100*round( annualfee/100) as rounded_af
    , annualfee
 from
     cls.cars;


  rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
            0                5           700          680
          200              198          1400         1386
         5000             5020        309900       309852
          400              366         18400        18413
         2500             2507        133700       133690
[...]
```

In the example above the column `annualfee/100` is a floating point type which does not take an additional argument in the function and instead just rounds to the nearest whole number. Since it's been divided by 100, this will return the number rounded to the nearest 100. We then multiply it against 100 to get the original scale.

- The second option is to cast, or change the variables type, in a few different ways:
  1. **Use the CAST function** We can use the `CAST` command in order to explicitly change the

type. The CAST command is a bit awkward syntactically, as can be seen below:

```
 select
     round(registrations, -2) as rounded_reg
    , registrations
    , round( cast( annualfee as int), -2) as rounded_af
    , annualfee
 from
     cls.cars;


  rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
            0                5           700          680
          200              198          1400         1386
         5000             5020        309900       309852
          400              366         18400        18413
         2500             2507        133700       133690
[...]
```

rather than using standard parameters, a more sentence like construction occurs.

2. **Conversion with `::`** We can use `::` to explicitly cast a variable from one type to another. **This is Postgres only!**

```
 select
     round(registrations, -2) as rounded_reg
    , registrations
    , round( annualfee::int, -2) as rounded_af
    , annualfee
 from
     cls.cars;


  rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
            0                5           700          680
          200              198          1400         1386
         5000             5020        309900       309852
          400              366         18400        18413
         2500             2507        133700       133690
[...]
```

3. **Implicit conversion:** While not possible in every situation, Postgres will implicitly convert between types when operators are applied. For example, if you multiply a float against an integer, the result will be a float:

```
select
    annualfee / 5.0 as af
from
    cls.cars;


     af
-------
  136
  277.2
61970.4
 3682.6
26738
[...]
```

In this case, `annualfee` has been converted from an integer to a float.

- There is a big "gotcha" when using implicit conversion – when doing it the database attempts to determine which type you want if you aren't careful you may end with an unanticipated result. Consider the following:
- Look at what following returns, given that there are 659 rows where registrations is equal to 5:

```
select registrations / 10 as calc
from cls.cars where registrations = 5;


  calc
------
     0
     0
     0
     0
     0
[...]
```

- Why is this occurring? Because the database sees a query which divides two integers and thus assumes that the result is also going to be an integer. Importantly – this isn't rounding, it is simply cutting off the value.
- We can use implicit methods of conversion in order to solve this. Consider the following:

```
select registrations *1.0 / 10 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

or

```
select registrations /10.0 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

or

```
select registrations::float /10 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

The first two solutions work because they introduce a number with a decimal component. When the database attempts to do math between decimals and integers it presumes that the answer is going to be decimal and we get the expected result. The third answer uses the "::" operator to convert the integer into a floating point number.

There is one important difference between the first two solutions and the final solution. The final solution converts the data into a floating point number, not a numeric type. As we will learn later, these are not equivalent and there can be strong reasons to prefer one data type over the other.

- Finally, we could use the CAST function in order to complete this operation:

```
select
    CAST( registrations as float)/10 as calc
from
    cls.cars
where
    registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```