Lecture 03 - Select

Jan 25

```
Select [ Projected Columns ]
From Table ...
    join to ...
Where ...
Group By Column List
Having [ where on grouped data ]
Ordr By [ Columns ]
```

Let’s use our table from last time

```
create table vote_by_county (
    id             serial primary key,
    year           int default 2021,
    state          text default '--',        -- irritatingly all upper case.
    state_uc       text default '--',
    state_po       varchar(2) default '--',     -- Incorrectly Named Column!
    county_name    text default '--',        -- irritatingly all upper case.
    county_name_uc text default '--',
    county_fips    int default 0,
    office         text default 'unk',
    candidate      text default 'unk',
    candidate_uc   text default 'unk',
    party          text default 'unk',
    candidatevotes int default 0,
    totalvotes     int default 0,
    version        int,
    vote_mode      text
);
```

Let’s just insert a few rows to see how insert works:

```
insert into vote_by_county  ( year, state, county_name, version ) values
    ( 2022, 'Wyoming', 'Albeny',   1 );
insert into vote_by_county  ( year, state, county_name, version ) values
    ( 2022, 'Wyoming', 'Big Horn', 2 );
insert into vote_by_county  ( year, state, county_name, version ) values
    ( 2022, 'Wyoming', 'Carbon',   8 );
```

and do some selects with the projected columns.

```
select id, year
 from vote_by_county
;
```

we can rename a column

```
select id, year as "Year of Our Lord"
 from vote_by_county
;
```

we can pick different columns

```
select id, year, state, county
    from vote_by_county
;
```

How about sorting the data

```
select id, year, state, county
    from vote_by_county
    order by county, state
;
```

you can only sort by the columns that you have in the *projected columns*.

you can use the column position

```
select id, year, state, county
    from vote_by_county
    order by 4, 3
;
```

you can ascending or descending sort

```
select id, year, state, county
    from vote_by_county
    order by 4 desc, 3 asc
;
```

You can apply functions and operators to the columns. In this case I will add 10 to the year and concatenate, `||` the state and county.

```
select id, year + 10 as "x", state||', '||county as "Location"
    from vote_by_county
    order by 4 desc, 3 asc
;
```

Single quotes denote string constants. Double quotes denote things like tables and column names. If you want an upper-lower case or a table name or column name with blanks then you have to quote it with double quotes.

Lot's of stuff will fail if you put blanks in your table names. This is worse than putting blanks in your file names. Python will crash with blanks in file names.

We will be using more than one table in queries. To do this we need to tell SQL the table. That is done with a table-alias. In this case *t1*.

```
select t1.id, t1.year + 10 as "x", t1.state||', '||t1.county as "Location"
    from vote_by_county  as t1
    order by 4 desc, 3 asc
;
```

There are lots of builtin functions that you can use and all sorts of arithmetic operators in PostgreSQL that can be applied to the projected columns. You can also write your own functions and pass data from the projected columns to the function and get back results.

Languages for functions include PG/SQL - the built in PostgreSQL language and others like JavaScript, Lua, C, C++, Go etc. I use PG/SQL and C for processing. JavaScript is 5 to 10x slower than PG/SQL. C is 10x faster than PG/SQL. These are rough numbers. I am a big fan of Go but I haven't used it for stored-procedure/functions yet in PostgreSQL.

Using a C function requires re-loading and re-starting the database - so it is hard.

Operators

https://www.postgresql.org/docs/9.0/functions.html

There are lots!

Operator	Description	Example	Result
`+`	addition	`2 + 3`	`5`
`-`	subtraction	`2 - 3`	`-1`
`*`	multiplication	`2 * 3`	`6`
`/`	division (integer division truncates the result)	`4 / 2`	`2`
`%`	modulo (remainder)	`5 % 4`	`1`
`^`	exponentiation	`2.0 ^ 3.0`	`8`
`|/`	square root	`|/ 25.0`	`5`
`||/`	cube root	`||/ 27.0`	`3`
`!`	factorial	`5 !`	`120`
`!!`	factorial (prefix operator)	`!! 5`	`120`
`@`	absolute value	`@ -5.0`	`5`
`&`	bitwise AND	`91 & 15`	`11`
`|`	bitwise OR	`32 | 3`	`35`
`#`	bitwise XOR	`17 # 5`	`20`
`~`	bitwise NOT	`~1`	`-2`
`<<`	bitwise shift left	`1 << 4`	`16`
`>>`	bitwise shift right	`8 >> 2`	`2`

and string operations

Function	Return Type	Description	Example	Result
`string || string`	`text`	String concatenation	`'Post' || 'greSQL'`	`PostgreSQL`
`string || non-string` or `non-string || string`	`text`	String concatenation with one non-string input	`'Value: ' || 42`	`Value: 42`
`bit_length(string)`	`int`	Number of bits in string	`bit_length('jose')`	`32`
`char_length(string)` or `character_length(string)`	`int`	Number of characters in string	`char_length('jose')`	`4`
`lower(string)`	`text`	Convert string to lower case	`lower('TOM')`	`tom`
`octet_length(string)`	`int`	Number of bytes in string	`octet_length('jose')`	`4`
`overlay(string placing string from int [for int])`	`text`	Replace substring	`overlay('Txxxxas' placing 'hom' from 2 for 4)`	`Thomas`
`position(substring in string)`	`int`	Location of specified substring	`position('om' in 'Thomas')`	`3`
`substring(string [from int] [for int])`	`text`	Extract substring	`substring('Thomas' from 2 for 3)`	`hom`
`substring(string from pattern)`	`text`	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '...$')`	`mas`
`substring(string from pattern for escape)`	`text`	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '%#"o_a#"_' for '#')`	`oma`
`trim([leading | trailing | both] [characters] from string)`	`text`	Remove the longest string containing only the `characters` (a space by default) from the start/end/both ends of the `string`	`trim(both 'x' from 'xTomxx')`	`Tom`
`upper(string)`	`text`	Convert string to upper case	`upper('tom')`	`TOM`

Base Functions

Function	Return Type	Description	Example	Result
`abs(x)`	(same as input)	absolute value	`abs(-17.4)`	`17.4`
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	`3`
`ceil(dp or numeric)`	(same as input)	smallest integer not less than argument	`ceil(-42.8)`	`-42`
`ceiling(dp or numeric)`	(same as input)	smallest integer not less than argument (alias for `ceil`)	`ceiling(-95.3)`	`-95`

Function	Return Type	Description	Example	Result
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	`28.6478897565412`
`div(y numeric, x numeric)`	`numeric`	integer quotient of `y`/`x`	`div(9,4)`	`2`
`exp(dp or numeric)`	(same as input)	exponential	`exp(1.0)`	`2.71828182845905`
`floor(dp or numeric)`	(same as input)	largest integer not greater than argument	`floor(-42.8)`	`-43`
`ln(dp or numeric)`	(same as input)	natural logarithm	`ln(2.0)`	`0.693147180559945`
`log(dp or numeric)`	(same as input)	base 10 logarithm	`log(100.0)`	`2`
`log(b numeric, x numeric)`	`numeric`	logarithm to base `b`	`log(2.0, 64.0)`	`6.0000000000`
`mod(y, x)`	(same as argument types)	remainder of `y`/`x`	`mod(9,4)`	`1`
`pi()`	`dp`	"π" constant	`pi()`	`3.14159265358979`
`power(a dp, b dp)`	`dp`	`a` raised to the power of `b`	`power(9.0, 3.0)`	`729`
`power(a numeric, b numeric)`	`numeric`	`a` raised to the power of `b`	`power(9.0, 3.0)`	`729`
`radians(dp)`	`dp`	degrees to radians	`radians(45.0)`	`0.785398163397448`
`round(dp or numeric)`	(same as input)	round to nearest integer	`round(42.4)`	`42`
`round(v numeric, s int)`	`numeric`	round to `s` decimal places	`round(42.4382, 2)`	`42.44`
`sign(dp or numeric)`	(same as input)	sign of the argument (-1, 0, +1)	`sign(-8.4)`	`-1`
`sqrt(dp or numeric)`	(same as input)	square root	`sqrt(2.0)`	`1.4142135623731`
`trunc(dp or numeric)`	(same as input)	truncate toward zero	`trunc(42.8)`	`42`
`trunc(v numeric, s int)`	`numeric`	truncate to `s` decimal places	`trunc(42.4382, 2)`	`42.43`
`width_bucket(op numeric, b1 numeric, b2 numeric, count int)`	`int`	return the bucket to which `operand` would be assigned in an equidepth histogram with `count` buckets, in the range `b1` to `b2`	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`
`width_bucket(op dp, b1 dp, b2 dp, count int)`	`int`	return the bucket to which `operand` would be assigned in an equidepth histogram with `count` buckets, in the range `b1` to `b2`	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`

How About the square root operator!

```
select |/25.0;
```

and a factorial operator!

```
select 5!;
```

and

```
select !! 5;
```

That is *FUN* ! Not 1 but 2 factorial operators.