>>

SQL Expressions

- Expressions in SQL
- SQL Operators
- The **`NULL`** Value
- Conditional Expressions

∧ >>

❖ Expressions in SQL

Expressions in SQL involve: objects, constants, operators

- objects are typically names of attributes (or PLpgSQL variables)
- operators may be symbols (e.g. **+**, **=**) or keywords (e.g. **between**)

SQL constants are similar to typical programming language constants

- integers: **123**, **-5**; floats: **3.14**, **1.0e-3**; boolean: **true**, **false**

But strings are substantially different

- **'...'** rather than **"..."**, no **\n**-like "escape" chars
- escape mechanisms: **'O''Brien'** or **E'O\'Brien'** (non-standard)
- dollar quoting: **$$O'Brien$$** or **$*tag*$O'Brien$*tag*$**

To escape chars, you can use the standard way of using a ' to escape the following char, or you can use postgreses custom way of doing it with the E prefix and then using \

❖ SQL Operators

Comparison operators are defined on all types:

`<` `>` `<=` `>=` `=` `<>`

In PostgreSQL, `!=` is a synonym for `<>` (but there's no `==`)

Boolean operators **AND**, **OR**, **NOT** are also available

Note **AND**,**OR** are not "short-circuit" in the same way as C's **&&**,**||**

Most data types also have type-specific operations available

String comparison (e.g. $str_1 < str_2$) uses dictionary order

See PostgreSQL Documentation Chapter 8/9 for data types and operators

❖ SQL Operators (cont)

SQL provides pattern matching for strings via **LIKE** and **NOT LIKE**

- **%** matches anything (cf. regexp **.***)
- **_** matches any single char (cf. regexp **.**)

Examples:

`name LIKE 'Ja%'`	`name` begins with 'Ja'
`name LIKE '_i%'`	`name` has 'i' as 2nd letter
`name LIKE '%o%o%'`	`name` contains two 'o's
`name LIKE '%ith'`	`name` ends with 'ith'
`name LIKE 'John'`	`name` equals 'John'

PostgreSQL also supports case-insensitive matching: **ILIKE**

❖ SQL Operators (cont)

PostgreSQL provides regexp-based pattern matching via `~` and `!~`

Examples (using POSIX regular expressions):

`name ~ '^Ja'`	`name` begins with 'Ja'
`name ~ '^.i'`	`name` has 'i' as 2nd letter
`name ~ '.*o.*o.*'`	`name` contains two 'o's
`name ~ 'ith$'`	`name` ends with 'ith'
`name ~ 'John'`	`name` contains 'John'

Also provides case-insensitive matching via `~*` and `!~*`

❖ SQL Operators (cont)

Other operators/functions for string manipulation:

- *str*$_1$ `||` *str*$_2$ ... return concatenation of *str*$_1$ and *str*$_2$
- **`lower(`***str***`)`** ... return lower-case version of *str*
- **`substring(`***str*,*start*,*count***`)`** ... extract substring from *str*

Etc. etc. ... consult your local SQL Manual (e.g. PostgreSQL Sec 9.4)

Note that above operations are null-preserving (strict):

- if any operand is **`NULL`**, result is **`NULL`**
- beware of **`(a||' '||b)`** ... **`NULL`** if either of **`a`** or **`b`** is **`NULL`**

<< ^ >>

❖ SQL Operators (cont)

Arithmetic operations:

```
+  -  *  /  abs  ceil  floor  power  sqrt  sin  etc.
```

Aggregations "summarize" a column of numbers in a relation:

- **count(***attr***)** ... number of rows in *attr* column
- **sum(***attr***)** ... sum of values for *attr*
- **avg(***attr***)** ... mean of values for *attr*
- **min/max(***attr***)** ... min/max of values for *attr*

Note: **count** applies to columns of non-numbers as well.

❖ The NULL Value

Expressions containing **NULL** generally yield **NULL**.

However, boolean expressions use three-valued logic:

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

❖ The `NULL` Value (cont)

Important consequence of **`NULL`** behaviour ...

These expressions do not work as (might be) expected:

```
x = NULL     x <> NULL
```

Both return **`NULL`** regardless of the value of *x*

Can only test for **`NULL`** using:

```
x IS NULL     x IS NOT NULL
```

❖ Conditional Expressions

Other ways that SQL provides for dealing with **`NULL`**:

`coalesce(`val_1, val_2, ...val_n**`)`**

- returns first non-null value val_i
- useful for providing a "displayable" value for nulls

E.g. **`select coalesce(mark,'??') from Marks ...`**

`nullif(`val_1, val_2**`)`**

- returns **`NULL`** if val_1 is equal to val_2
- can be used to implement an "inverse" to **`coalesce`**

E.g. **`nullif(mark,'??')`**

<< ∧

❖ Conditional Expressions (cont)

SQL also provides a generalised conditional expression:

```
CASE
    WHEN test₁ THEN result₁
    WHEN test₂ THEN result₂
    ...
    ELSE resultₙ
END
```

E.g. **`case when mark>=85 then 'HD'`** **...** **`else '??' end`**

Tests that yield **`NULL`** are treated as **`FALSE`**

If no **`ELSE`**, and all tests fail, **`CASE`** yields **`NULL`**

Produced: 22 Sep 2020