Disclaimer: After the first few examples, the "**SELECT** column_name **FROM** table_name" will be omitted, leaving only the relevant clauses.

Lesson 4: Filtering Data

WHERE clause

- Used for specifying *search criteria* a.k.a. *filter condition*.
- Why? Often we only want to work with a specific subset of rows.
- Example:

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price = 3.49;
```

- Example; Checking for nonmatches:

```
SELECT vend_id, prod_name
FROM Products
WHERE vend_id <> 'DLL01';
```

- Usual logical/comparison operators are available: = , != (or <>) , < , <= , !< , > , >= , !>

Note: SQL equality syntax doesn't correspond with what we're accustomed to in the more widely used programming languages. Typically "=" is the assignment operator and "==" is the comparison operator used for checking equality.

TIP: SQL Versus Application Filtering

- Data can also be filtered at the client application level, instead of by the DBMS.
- Strongly discouraged--against best practices!
- Why? Waste of bandwidth, bogs down the client, results in bloated client code, etc.
- Databases are optimized to perform filtering quickly and efficiently.
- Recall that we always want to use the DBMS for what it's best at (joins, filtering, sorting).

CAUTION: **WHERE** Clause Position; **ORDER BY** always after **WHERE**:

```
SELECT
FROM
WHERE
ORDER BY
```

Note: I'm not going to insert one here, but I would suggest newer folks do an image search for "SQL order of clauses" or "SQL order of operations"--snag your favorite for your notes.

TIP: When to Use Quotes

- DO use single quotes for string data types (**CHAR**, **VARCHAR**, **ENUM**, ...).
- DO use single quotes for date/time data types (**DATE**, **DATETIME**, **TIMESTAMP**, ...).
- DON'T use quotes for numeric types (**BOOL**, **INT**, **FLOAT**, **DOUBLE**, ...).
- Double quotes typically reserved for encapsulating strings that contain single quotes, or sometimes used when referring to certain identifiers/objects (check your DBMS doc).

BETWEEN operator

- Used with the **WHERE** clause to return values within a range.
- Interval endpoints are inclusive.
- Syntax: **WHERE** column_name **BETWEEN** value_1 **AND** value_2
- Basically just syntactic sugar; for convenience.
- Can be used with strings and datetimes--careful though, lots of 'gotchas'.
- Example:

```
WHERE prod_price BETWEEN 5 AND 10;
```

NULL

- No value, as opposed to a field containing 0, or an empty string, or just spaces.
- Use **IS NULL** clause to check for existence of **NULL** values, not " = **NULL**".

```
WHERE prod_price IS NULL;
```

- Rows with **NULL** in the filter column not returned when filtering for matches/nonmatches.
  Spoiler: Nulls are not included in aggregate functions (**AVG()**, **COUNT()**, ...).
- Example; Can check if a col is null even if you're returning values from another col:

```
SELECT cust_name
FROM Customers
WHERE cust_email IS NULL;
```

- Many DBMSs extend the standard set of operators--check documentation!
  Luckily, despite some variance (**ISNULL()**, **IFNULL()**, **NVL()**), all the major DBMS seem to support **COALESCE()**.

Lesson 5: Advanced Data Filtering

TL;DR: You can do more advanced filtering in the **WHERE** clause by chaining an arbitrary number of conditions with **AND** or **OR** operators. Also, the **IN** and **NOT** operators can be used with your conditions, mostly for convenience/readability.

Nice little table from sqlbolt.com:

Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical operators	col_name != 4
BETWEEN ... AND ...	Number is within range of two values (inclusive)	col_name **BETWEEN** 1.5 **AND** 10.5
NOT BETWEEN ... AND ...	Number is not within range of two values (inclusive)	col_name **NOT BETWEEN** 1 **AND** 10
IN (...)	Number exists in a list	col_name **IN** (2, 4, 6)
NOT IN (...)	Number does not exist in a list	col_name **NOT IN** (1, 3, 5)

AND Operator

- Returns rows that match both conditions.

- Why? To filter by more than one column or condition i.e. more granular searching.
- Syntax: **WHERE** condition_1 **AND** condition_2
- Example:

    WHERE vend_id = 'DLL01' **AND** prod_price <= 4;
- Additional filtering via chaining; each condition separated by an AND keyword.

    WHERE condition_1 **AND** condition_2 **AND** condition_3 ...;
- As dictated by many style guides, it's common to see **AND**s/**OR**s on separate lines:

    WHERE condition_1
    AND condition_2
    ...
    AND condition_n;

OR Operator

- Retrieves rows that match either condition.
- This operator short-circuits, just like in a normal programming language.

    DBMSs will not evaluate the second condition if the first condition is satisfied.
    i.e. this is a normal logical **OR**, not an exclusive **OR**, otherwise known as **XOR**.
- Example:

    WHERE vend_id = 'DLL01' **OR** vend_id = 'BRS01';

Chaining multiple conditions with ANDs and ORs

- SQL (like most languages) processes **AND** operators before **OR** operators.
- Use parentheses to explicitly group related operators.
- Parentheses have a higher order of evaluation than either **AND** or **OR** operators.

    As mentioned previously, always opt to be explicit!
- The two following examples will give different results!

    WHERE vend_id = 'DLL01' **OR** vend_id = 'BRS01'
    AND prod_price >= 10;

    WHERE (vend_id = 'DLL01' **OR** vend_id = 'BRS01')
    AND prod_price >= 10;

IN operator

- Used to specify a range of conditions.
- Takes a comma-delimited list of valid values.
- These two are equivalent:

    WHERE vend_id **IN** ('DLL01', 'BRS01')
    WHERE vend_id = 'DLL01' **OR** vend_id = 'BRS01'
- Allows for more succinct/readable queries.
    - IN (val1, val2, val3, ... , val4023994358) v.s. a buttload of **OR** statements.
- **IN** operators almost always executes more quickly than lists of **OR** operators.
- Spoiler: **IN** operator can contain another **SELECT** statement, enabling highly dynamic **WHERE** clauses.

NOT operator

- Negates the following condition.
- These two are equivalent:

```
WHERE NOT vend_id = 'DLL01'
WHERE vend_id <> 'DLL01'
```

- **NOT** works well in conjunction with an **IN** operator to simplify queries.