Useful tools

Often, SQL result sets are made to examine a subset of data. Here are two handy tools we can use to make them readable, and save results for future use.

We often draw upon different databases and tables which may have complex column names. The SQL keyword **AS** is a handy tool to rename columns in a query **result**.

At times, it is useful to save results so we can access them later. **`INSERT INTO`** can be used to save a query as well. Instead of specifying **`VALUES`**, we use a query.

Useful tools in SQL

Aliasing

> **Aliasing** is a technique that uses the **AS** keyword to **temporarily rename** a table or column in a query for simplification and readability.

Syntax: `SELECT col_name_in_db AS new_col_name FROM db.table;`

Table_2

col1	col2	col3
z	68	s
x	1	l
w	7	l
y	56	m

```
SELECT
      col1 AS Initials
      col2,
      col3,
FROM
      db.Table_2;
```

Results

Initials	col2	col3
z	68	s
x	1	l
w	7	l
y	56	m

Data Query Language

Aliasing

You cannot rename a column with **AS** and use the alias in a **`WHERE`** clause.

AS creates an **alias** or shortcut name for a column **in a SQL results set**. Renaming columns in a results set makes the table **more readable**.

Always choose **descriptive names** such as *first_name*, *last_name* instead of *name*, and *is_subscribed* instead of *sub*.

Useful tools in SQL

Saving a results set in a table

> To save useful results as a table, we can use **`INSERT INTO`** to save the results of a query to an **existing** table.

Syntax: `INSERT INTO table_name(col_name(s)) SELECT col(s) FROM db_name.table_name;`

Table_2

col1	col2	col3
z	68	s
x	1	l
w	7	l
y	56	m

```
CREATE TABLE
      Saved_table(
          Initials VARCHAR(255),
          Cost INT
      );
INSERT INTO
    Saved_table(
        Initials,
        Cost
    )
SELECT
      col1,
      col2
FROM
      db.Table_2;
```

Saved_table

Initials	Cost
z	68
x	1
w	7
y	56

Useful tools in SQL

Saving a results set in a table

```
CREATE TABLE
      Saved_table(
      Initials VARCHAR(255),
      Cost INT
      );
```

```
INSERT INTO
      Saved_table(
      Initials,
      Cost
      )
```

```
SELECT
      col1,
      col2
FROM
      db.Table_2;
```

This involves three steps:

1. **Create a new table** called `Saved_table`, with `Initials` and `Cost` as the column names.
2. **Query** the columns/data we want to save (**`SELECT`** `col1` and `col2`).
3. **Insert the results** of the query (`col1` and `col2`) into the new table (`Initials` and `Cost`) – **`INSERT INTO`**.

Comparison operators

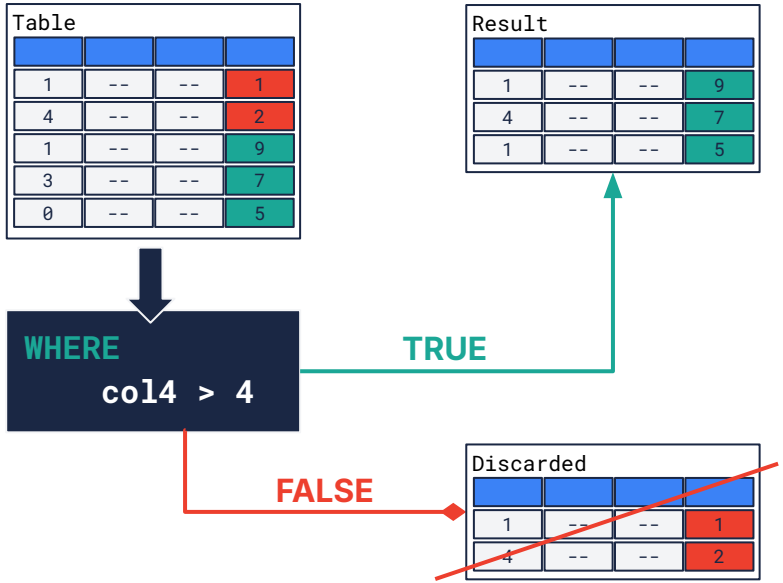
SQL operators

> Using the **`WHERE`** keyword, we can filter data. **Operators** can be used in conjunction with the **`WHERE`** clause to specify filter conditions.

Practical uses:

- Find all countries with a gross domestic product (GDP) of **more than** $400 billion.
- Find global regions with a population of **less than** 1 million.
- Find the development indexes **after** a certain date **and** for a **specific** country.
- Find elder death rates for sub-Saharan countries during the COVID-19 crisis (**between** 2020 and 2023).

For example, a **greater than** operator (**>**) in SQL only returns rows that have a value larger than 4 in `col4`.



Comparison operators

> Comparison operators are used to **compare values** and determine the truth or falsity of a condition based on the comparison result.

Examples of comparison operators in SQL include:

- equals (**=**)
- not equals (**<>**)
- less than (**<**)
- greater than (**>**)
- less than or equal to (**<=**)
- greater than or equal to (**>=**)

A square is a triangle → **FALSE**

A circle is not a triangle → **TRUE**

23 > 1

23 is greater than 1 → **TRUE**

“Kenya” = “Nigeria”

The strings “Kenya” and “Nigeria” are the same → **FALSE**

Comparison operators

> In SQL, comparison operators are commands used with **`WHERE`** that return the data in a row if the condition is **TRUE**.

=	>	>=	<	<=	<>
Checks if **A = B** Returns the rows where the value **A** in a column **equals** a specified value **B**.	Checks if **A > B** Returns the rows where value **A** in a column is **greater than** a specified value **B**.	Checks if **A ≥ B** Returns the rows where value **A** in a column is **greater than or equal to** a specified value **B**.	Checks if **A < B** Returns the rows where value **A** in a column is **less than** a specified value **B**.	Checks if **A < B** or if **A ≤ B** Returns the rows where value **A** in a column is **less than or equal to** a specified value **B**.	Checks if **A ≠ B** Returns the rows where value **A** in a column is **not equal** to a specified value **B**.

Our example database

We will use this placeholder database as an example:

Database (db)

Table_1

col1	col2	col3
x	34	s
y	73	m
z	22	l
w	12	m

Table_2

col1	col2	col3
car	68	s
cat	1	l
pet	7	l
cart	56	m

Table_3

col1	col2	col3
x	NULL	s
x	42	s
z	NULL	s
x	7	s

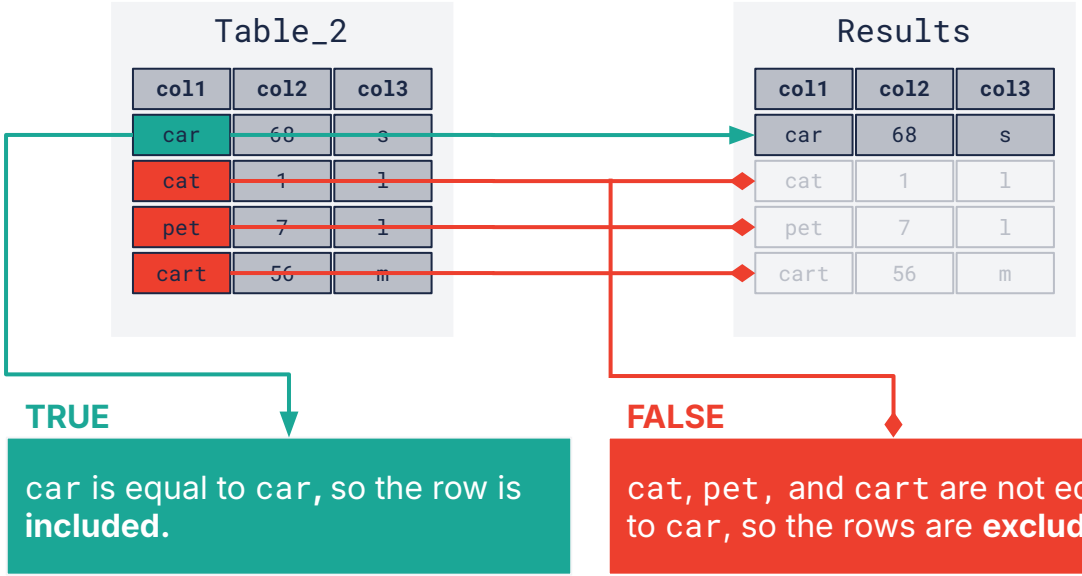
Note: These are bad naming examples, but we use them here to show how queries work. Always try to use more descriptive titles.

Comparison operators

Equal to

> The = operator returns the rows where the value in a column **equals** a specified value.

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col1 = "car";
```



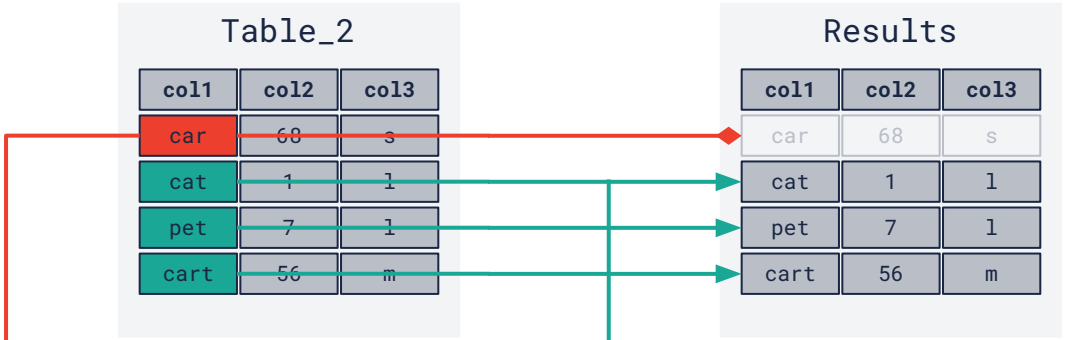
Note that **car** is a **string**, so we enclose it in quotes.

Comparison operators

Not equal to

> The operator `!=` or `<>` can be used to return the rows where the value in a column is **not equal to** a specified value. Both `!=` and `<>` serve as alternatives for expressing inequality.

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col1 <> "car";
```



FALSE

`car` is equal to `car`, so the row is **excluded.**

TRUE

`cat`, `pet`, and `cart` are **not** equal to `car`, so the rows are **included.**

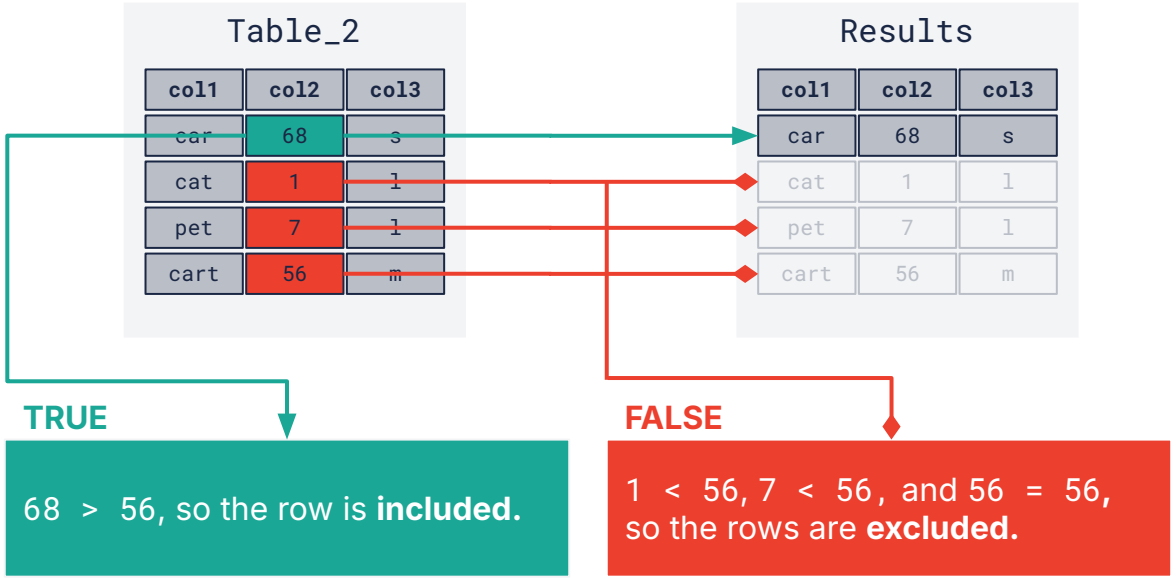
Note that we add a space before and after the operator in SQL code to make it more readable.

Comparison operators

Greater than

The `>` operator returns the rows where the value in a column is **greater than** a specified value.

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col2 > 56;
```

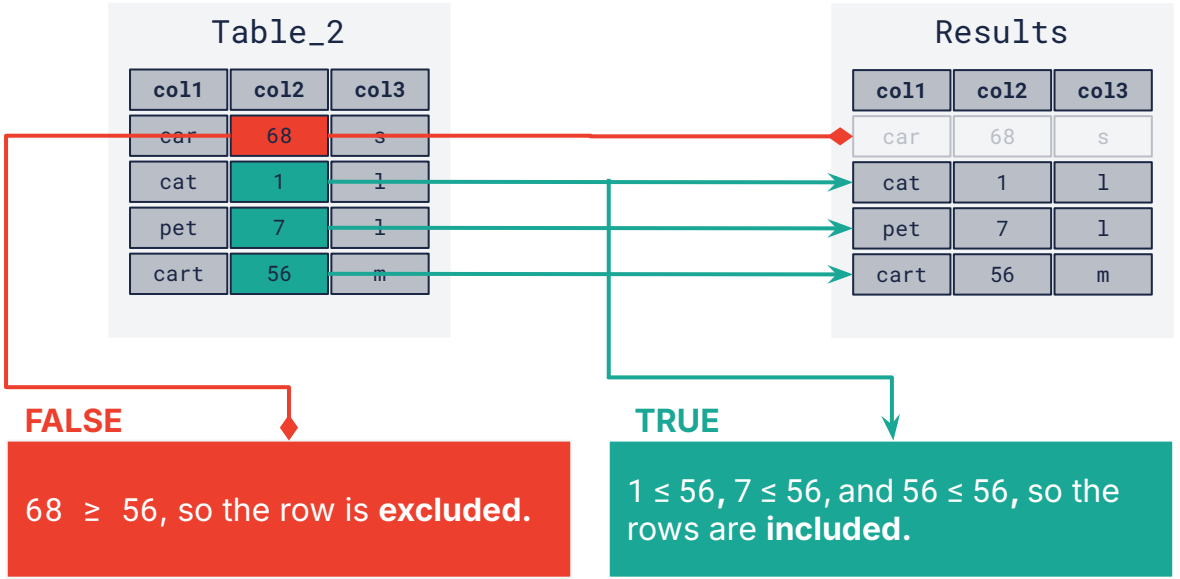


Comparison operators

Less than or equal to

The `<=` operator returns the rows where the value in a column is **less than or equal to** a specified value.

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col2 <= 56;
```



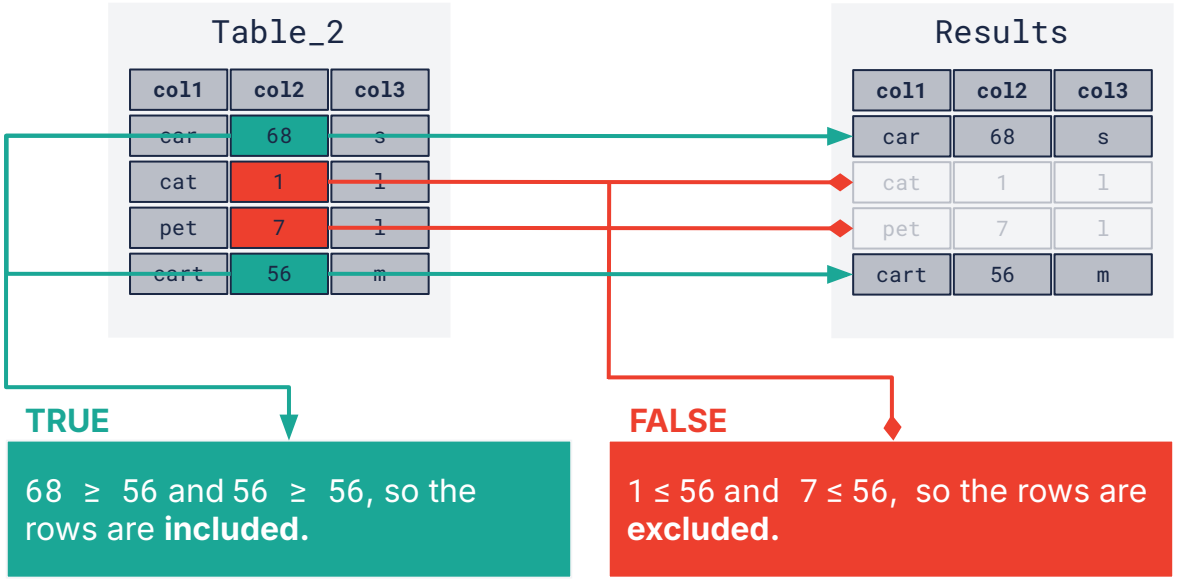
Using `col2 <= 56` includes 56, unlike `<`.

Comparison operators

Greater than or equal to

The `>=` operator returns the rows where the value in a column is **greater than or equal to** a specified value.

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col2 >= 56;
```



Logic operators

> Logic (or boolean) operators **combine**, **exclude**, or **negate conditions** in order to evaluate the overall truth of a condition or a set of conditions.

`AND` combines two conditions and is only **TRUE** if **both** conditions are **TRUE**.

`OR` combines two conditions and is only **TRUE** if **either** condition is **TRUE**.

`NOT` reverses the truth of a condition. **TRUE** becomes **FALSE** and **FALSE** becomes **TRUE**.

`IN` combines several **`OR`** operators. It returns **TRUE** if a value is within a list of possible values.

`BETWEEN` combines the **`>`** and **`<`** operators. It returns **TRUE** if a value is within a specified range.

`LIKE` matches a string to a pattern. It returns **TRUE** if a string matches the search pattern.

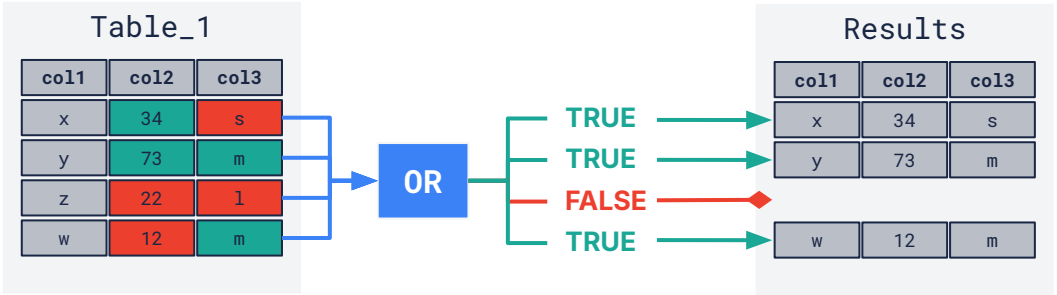
Logic operators

OR

> The **OR** operator is used to filter records based on multiple conditions. If **at least one of the specified conditions** is **TRUE**, the record will be included in the results set.

Syntax: `… WHERE condition1 OR condition2;`

```
SELECT
    *
FROM
    db.Table_1
WHERE
    col2 >= 25
    OR col3 = “m”;
```



Rows that meet **any** of the conditions (`col2 ≥ 25` **or** `col3 = “m”`) are **included**.

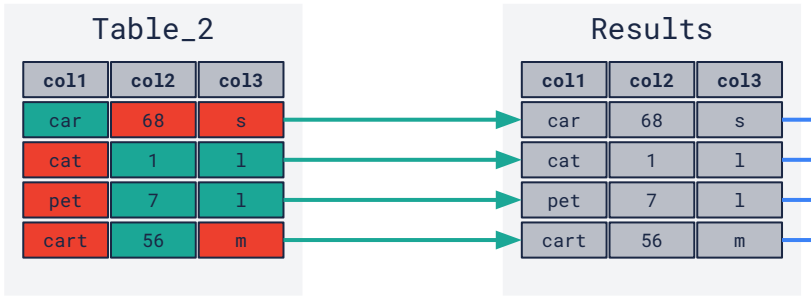
Logic operators

Multiple OR conditions

OR

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col1 = "car"    -- Condition 1
    OR col2 < 60    -- Condition 2
    OR col3 = "1"; -- Condition 3
```

- More than two **OR** statements can be combined.
- Rows that meet **any** of the specified conditions are included.
- Rows are **included** in the results if `col1 = car` **or** when `col2 > 60` **or** `col3 = 1`.



`car` matches the first condition, so the row is **included**.

`1` and `1` meet conditions 2 and 3, so the row is **included**.

`7` and `1` meet conditions 2 and 3, so the row is **included**.

`56` meets condition 2, so the row is **included**.

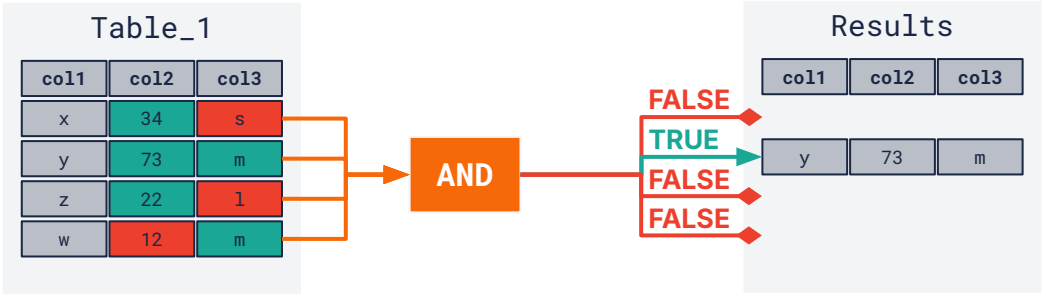
Logic operators

AND

> The **AND** operator is used to filter records based on **more than one condition**. **All** conditions connected by an **AND** clause must be **TRUE** for the record to be included in the results.

Syntax: `… WHERE condition1 AND condition2;`

```
SELECT
    *
FROM
    db.Table_1
WHERE
    col2 >= 20
    AND col3 = "m";
```



Only rows that are **TRUE** for **both** conditions (`col2 ≥ 20 AND col3 = "m"`) are included.

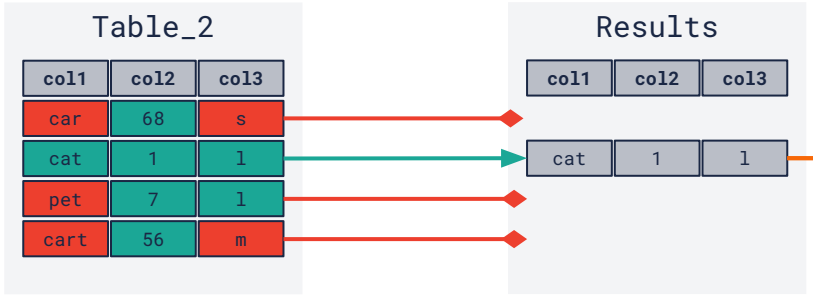
Logic operators

Multiple AND conditions

AND

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col1 = "cat"      -- Condition 1
    AND col2 > 0      -- Condition 2
    AND col3 = "1";   -- Condition 3
```

- More than two **AND** statements can be combined.
- Rows that meet **all** of the conditions are included.
- Rows are only **included** in the results if `col1 = cat` **and** `col2 > 0` **and** `col3 = 1`.



Only one row meets **all conditions**.

`cat` meets condition 1
AND `1` meets condition 2,
AND `1` meets condition 3, so the row is **included**.

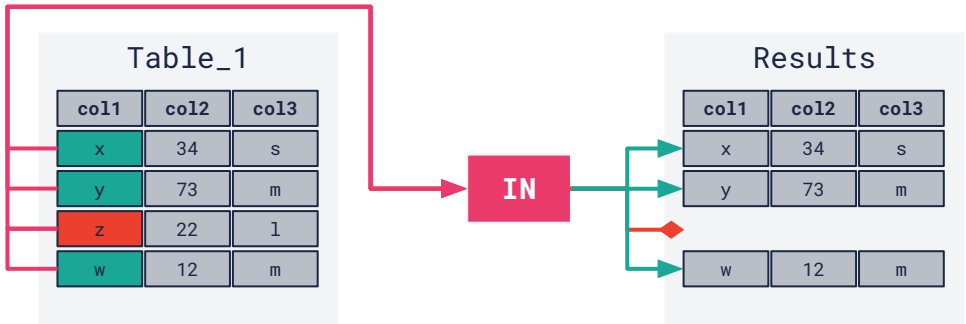
Logic operators

IN

> **IN** is used to check if a value in a column matches any value in a list.

Syntax: `… WHERE col IN (value1, value2, ...);`

```
SELECT
    *
FROM
    db.Table_1
WHERE
    col1
IN(
    “w”,
    “x”,
    “y”
);
```



col1 **IN**(“w”, “x”, ”y”) is a shortcut for: (col1 = “x” **OR** col1 = “y” **OR** col1 = “w”).

It is better to use **IN** when checking multiple **OR** statements.

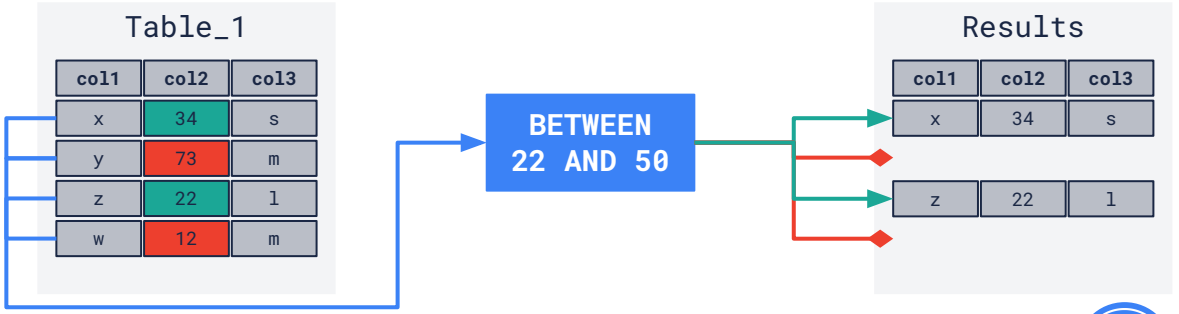
Logic operators

BETWEEN

> The **BETWEEN** operator is used to filter records within a specific **range**, **inclusive** of the range endpoints.

Syntax: `… WHERE col BETWEEN value1 AND value2;`

```sql
SELECT
    *
FROM
    db.Table_1
WHERE
    col2 BETWEEN 22 AND 50;
```



- Rows where `col2` is between `22` and `50` are **included**.
- Rows where `col2` is outside this range are **excluded**.

BETWEEN makes SQL code more readable, so always try to use it when specifying ranges.

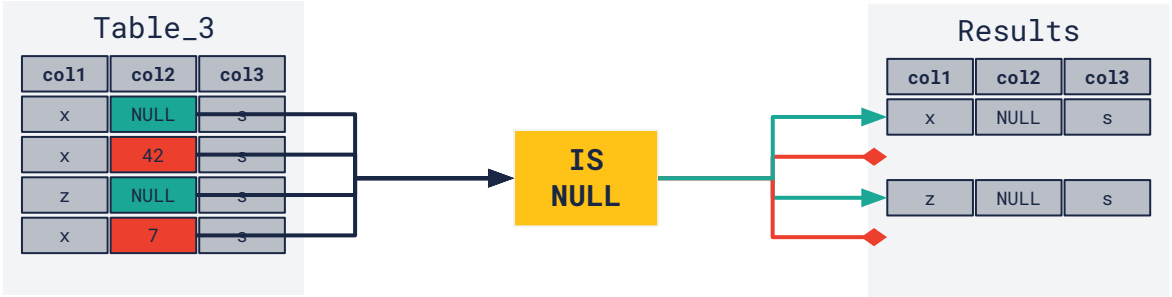
Logic operators

IS NULL

> **`IS NULL`** is used to check whether a value is **NULL** or **missing**, essentially helping to identify gaps in the data.

Syntax: `… WHERE col IS NULL;`

```
SELECT
    *
FROM
    db.Table_3
WHERE
    col2 IS NULL;
```



- **Includes** only rows where there are **NULL** values in the specified column.
- To check multiple columns for **NULL** values, we can use "**OR** `col3 IS NULL`" etc.

NULL values often create fallacies, so it is best to know about any **NULL** values in a column.

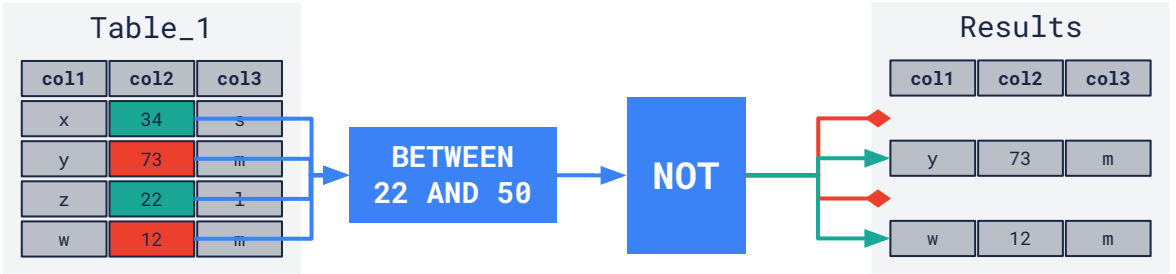
Logic operators

NOT and BETWEEN

> **NOT** is used to negate a condition. **NOT BETWEEN**, for example, excludes a specific range of values.

Syntax: `… WHERE col NOT BETWEEN value1 AND value2;`

```
SELECT
    *
FROM
    db.Table_1
WHERE
    col2 NOT BETWEEN 22 AND 50;
```



- 34 and 22 both satisfy the **BETWEEN** condition, and **NOT** reverses the outcome, so 34 and 22 are now **FALSE**, and those rows are excluded.
- 12 and 73 evaluate to **FALSE** in the **BETWEEN** condition, and are reversed by **NOT** to **TRUE**, so those rows are **included**.

NOT complicates SQL logic, so the code becomes less readable. Use **NOT** sparingly.

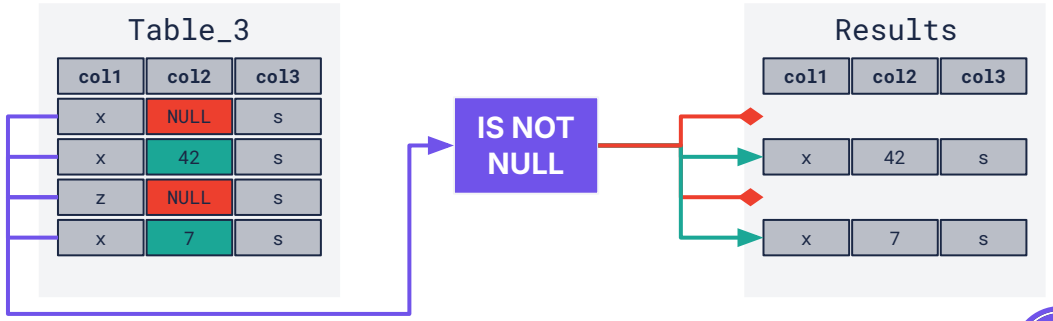
Logic operators

IS NOT NULL

> The **`IS NOT NULL`** operator checks to see if a value is not null/empty, helping to confirm when data do indeed exist.

Syntax: `… WHERE col IS NOT NULL;`

```
SELECT
    *
FROM
    db.Table_3
WHERE
    col2 IS NOT NULL;
```



Includes only rows where there are **no NULL** values in the specified column.

We can use **`IS NOT NULL`** to remove any rows with missing data.

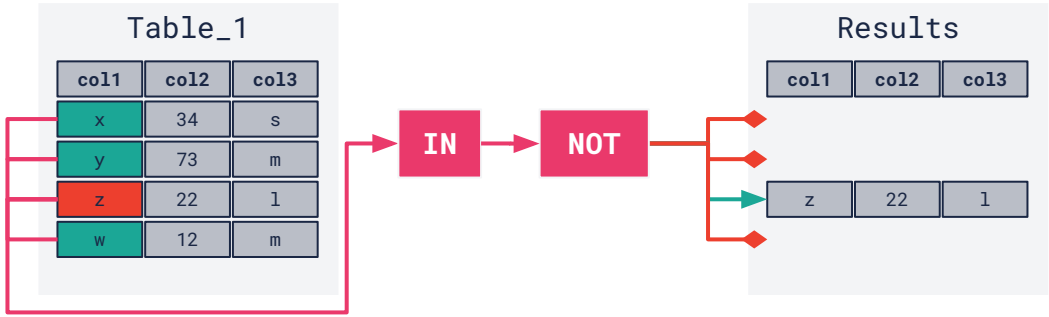
Logic operators

NOT and IN

> **`NOT IN`** is used to ensure a value **does not** match any value in a list. The outcome of **`IN`** is reversed by **`NOT`**.

Syntax: `… WHERE col NOT IN (value1, value2, ...);`

```sql
SELECT
    *
FROM
    db.Table_1
WHERE
    col1
IN(
    "w",
    "x",
    "y"
);
```



- **`NOT IN`** reverses **`IN`**, so rows where `col1 = (w, x, y)` are **excluded**.
- `z` is **`NOT IN`** the list of options, so the row is **included**.

Logic operators

SQL text searching

Databases house an overwhelming amount of **text-based data**, including names, addresses, descriptions, and categories.

The `LIKE` operator in SQL is our key tool for navigating this textual labyrinth, allowing targeted **searches** within this data using **wildcards** to tune our searches.

For instance, a humanitarian aid worker could use it to quickly locate all NGOs with names that are related to water within a massive database using `LIKE`.

Logic operators

Searching text in SQL

> `LIKE` is used in a `WHERE` clause to **search** for a specified pattern in a **text-based** column. These patterns can be expressed using **wildcards**.

Wildcards are symbols that can represent any character(s) `(a-z, A-Z, 0-9)`, and even symbols, enabling a pattern-based search with the **`LIKE`** operator. There are two wildcards in SQL – underscore `(_)` and percentage `(%)`.

Underscore (_)

Represents **a single** character.

A search pattern like `h_t` will match with values like `hot`, `hat` and `hit`, but would not match with `heat` because `_` specifies a single character.

Percentage (%)

Represents **multiple** characters.

A search pattern like `South%` will match with values like `South Korea`, `South Africa`, `Southern`, or `Southern#1594` since it can represent any number of characters.

Logic operators

Wildcards

> The **placement** of wildcards in the search pattern provides even more search flexibility.

`%` at the end: Matches any string starting with the given characters, for example, `p%` must **start** with `p`, be any length, and can end with any character.

`%` at the start: Matches any string ending with the given characters, for example, `%t` can start with any character, can be any length, but must **end** with `t`.

`_` in place of one character: Matches any **single** character in that position, for example, `_at` must contain **only three characters** and **end** with `t`.

	p%	%t	_at
car	FALSE	FALSE	FALSE
cat	FALSE	TRUE	TRUE
pet	TRUE	TRUE	FALSE
pat	TRUE	TRUE	TRUE
cart	FALSE	TRUE	FALSE

Logic operators

Wildcards

`%` **inside:** Matches any string that begins and ends with the given characters, and can be any length, for example, `c%t` must **start** with `c`, can contain any number of characters, and must **end** with `t`.

Wildcards can be combined:

`_` **at both ends:** Matches any string containing the given characters, three characters long, for example, `_a_` must be **three characters long** and can start and end with **any** character, but **must have** an `a` in the middle.

`%` **and** `_`**:** Using both `%` and `_` we can limit strings further, for example, `_a%` matches with cat and cart. `_a%` can start with any single character that must be followed by an `a` and can end with any number of characters.

	c%t	_a_	_a%
car	FALSE	TRUE	TRUE
cat	TRUE	TRUE	TRUE
pet	FALSE	FALSE	FALSE
pat	FALSE	TRUE	TRUE
cart	TRUE	FALSE	TRUE

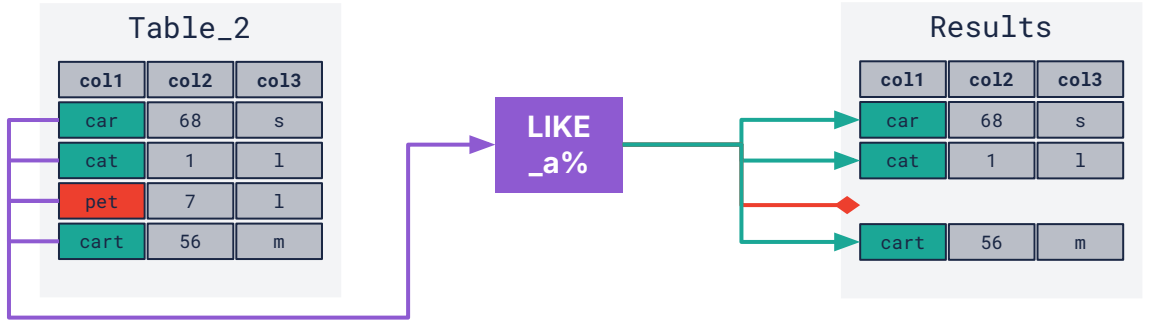
Logic operators

LIKE

> Only rows that **match** the `LIKE` search pattern (in the specified column) are **included** in the results.

Syntax: `… WHERE col LIKE “pattern + wildcard”;`

```
SELECT
    *
FROM
    db.Table_2
WHERE
    col1 LIKE “_a%”;
```



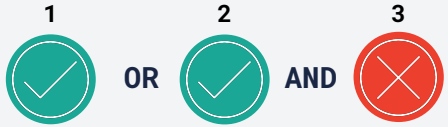
Only rows that **match** the search pattern (in the specified column) are **included** in the results.

- `pet` does not contain an `a`, so the row is **excluded**.
- `car`, `cat` and `cart` match `_a%` because `%` can be `r`, `t`, or `rt`.

Order of operations

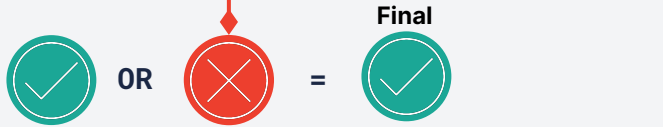
> Operations in parentheses `()` are evaluated first, then **`AND`**, and lastly, **`OR`** is evaluated.

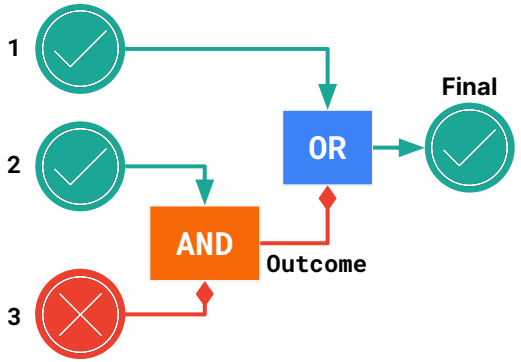
Suppose we have this combination of conditions:



There are no `()`, so we evaluate **`AND`** first:

Then we evaluate **`OR`** using the outcome of **`AND`**:





Keep the order of operations in mind when using **`AND`** and **`OR`** together.

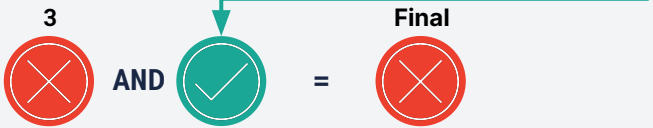
Order of operations

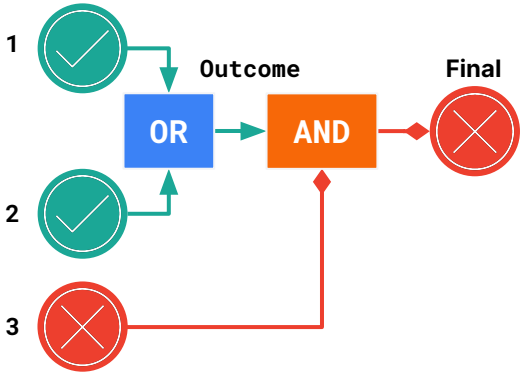
> Parentheses ( ) can **interrupt** the order of operations.

Suppose we have this combination of conditions:

We evaluate the OR inside the ( ) first:

Then we evaluate AND using the outcome of OR:



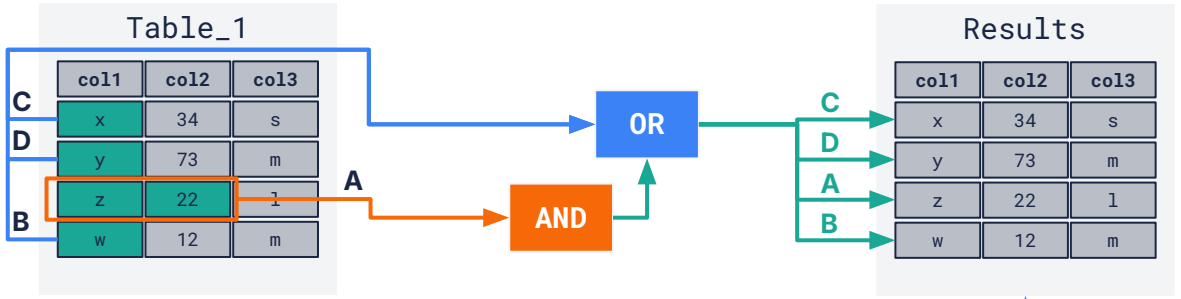


By using parentheses, we can alter the order in which conditions are checked. Using this we **create complex logic** in SQL to search for data using `WHERE`.

Logic operators

Order of operations using WHERE

```
SELECT
    *
FROM
    db.Table_1
WHERE
    col1 = "w"
    OR col1 = "x"
    OR col1 = "y"
    OR col1 = "z"
    AND col2 > 20;
```



SQL will execute **AND** first, then **OR**, so the query that SQL executes is:

```
…
WHERE
    col1 = "z"  AND  col2 > 20
     OR  col1 = "w"
     OR  col1 = "x"
     OR  col1 = "y";
```

This condition includes a row where: `col1 = z` **AND** `col2 > 20` is **TRUE**. — A

This condition includes a row where: `col1 = w` is **TRUE**. — B

This condition includes a row when: `col1 = x` is **TRUE**. — C

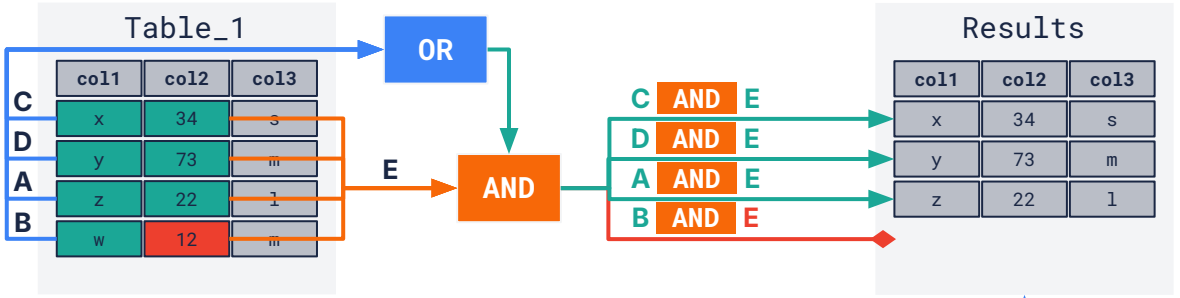
This condition includes a row when: `col1 = y` is **TRUE**. — D

A row is included if it is **TRUE** for:
A OR B OR C OR D

Logic operators

Using parentheses with WHERE

```
SELECT
    *
FROM
    db.Table_1
WHERE (
    col1 = "w"
    OR col1 = "x"
    OR col1 = "y"
    OR col1 = "z"
    )
    AND col2 > 20;
```



SQL will evaluate the contents of ( ) first, then **AND** :

```
…
WHERE
    (col1 = "z"
     OR col1 = "w"
     OR col1 = "x"
     OR col1 = "y")
     AND col2 > 20 ;
```

This condition includes a row where: `col1 = z` is **TRUE**. — A

This condition includes a row where: `col1 = w` is **TRUE**. — B

This condition includes a row when: `col1 = x` is **TRUE**. — C

This condition includes a row when: `col1 = y` is **TRUE**. — D

This condition includes a row when: `col2 > 20` is **TRUE**. — E

A row is only included if both:
(A, B, C, OR D) are **TRUE**
AND E is also **TRUE**.