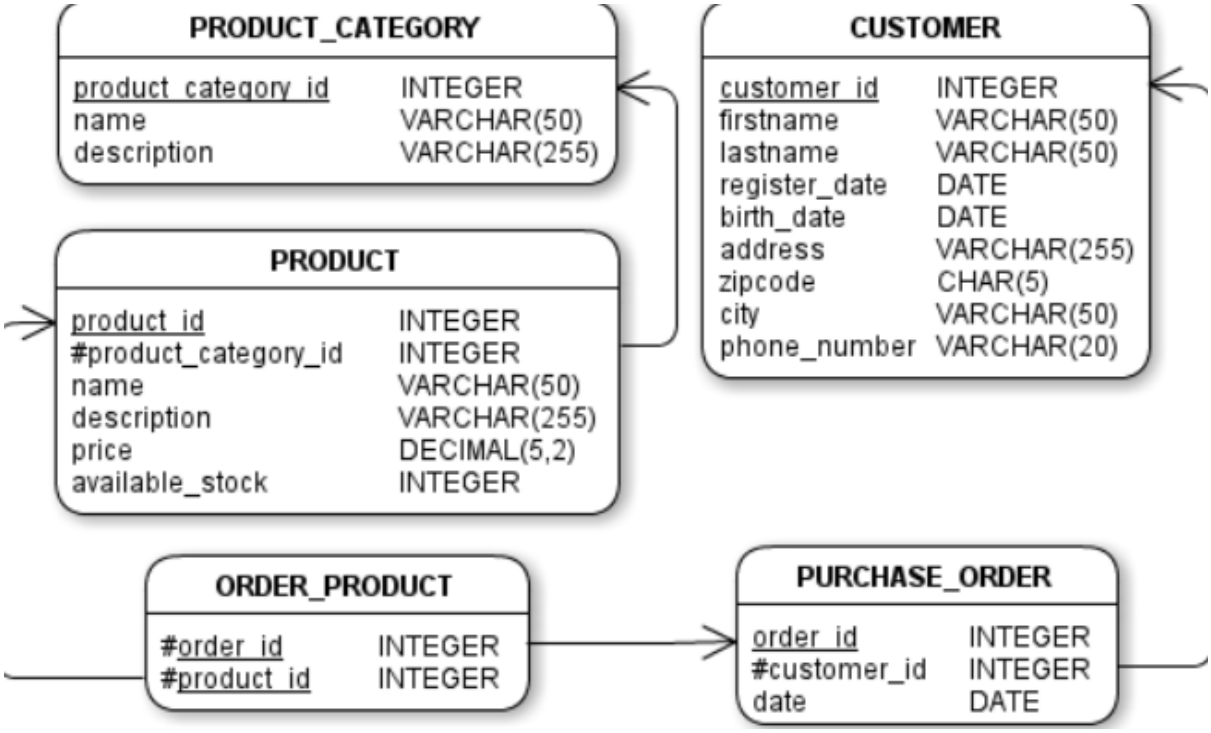
SQL Pathway (CodinGame)

The ERD for the database we will be using for most examples is shown below:



▼ Select Statements

Contains two statements, a `SELECT` clause and a `FROM` clause

```
SELECT column -- SELECT clause
FROM table;   -- FROM clause
```

▼ Select Statement Sequences

The clauses of a SQL query, (a.k.a. a Select Statement) must be written in a set order, known as the Logical Sequence or the Syntax Sequence.

▼ Syntax Sequence

- `SELECT`
- `DISTINCT`

- `FROM`
- `WHERE`
- `GROUP BY`
- `HAVING`
- `ORDER BY`

A contrived example, showing all of these clauses in the correct order, is shown in the code block below:

```
-- SQL request(s) below
SELECT DISTINCT pc.name, AVG(pr.price) AS "Average Price"
FROM product pr
    JOIN product_category pc
    ON pr.product_category_id = pc.product_category_id
WHERE pr.name LIKE '%0%'
GROUP BY pc.product_category_id
HAVING AVG(pr.price) > 100
ORDER BY pc.name ASC
```

⚠️ Syntax Sequence ≠ Processing Sequence

▼ Processing Sequence

- `FROM`
- `WHERE`
- `GROUP BY`
- `HAVING`
- `SELECT`
- `DISTINCT`
- `ORDER BY`

▼ `SELECT` clause and aliases

Specifies which columns we want the query to return data from e.g.,

```
SELECT column1, column2, column3
FROM table1
```

To return all columns in a table, use an asterisk *:

```
SELECT *
FROM customer;
```

▼ Aliases

- Used to improve presentation/readability.
- Good practice to use the `AS` keyword for column aliases, to help separate the alias from the column definition.
- Use double quotes `""`.

```
SELECT phone_number AS "Phone Number"
FROM customer
```

```
SELECT firstname AS "First Name", lastname AS "Last Name", birth_date AS "D
ate of Birth"
FROM customer
```

▼ `DISTINCT` clause

Removes duplicate rows from queries

```
SELECT DISTINCT lastname
FROM customer;
```

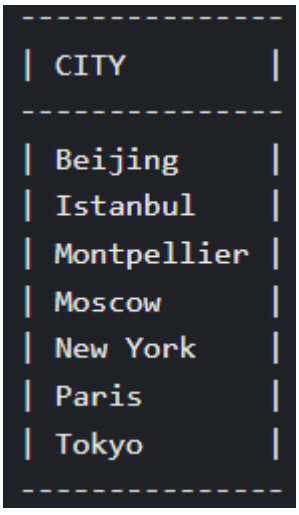
▼ Exercise: `DISTINCT` clause

The customer table contains the following columns:

- `customer_id`
- `firstname`
- `lastname`
- `register_date`
- `birth_date`
- `address`
- `zipcode`
- `city`
- `phone_number`

Return a list of cities we have customers for. Each city should only appear once in the results. Display the cities in alphabetical order.

```
SELECT DISTINCT city
FROM customer
ORDER BY city ASC;
```



```
---------------
| CITY        |
---------------
| Beijing     |
| Istanbul    |
| Montpellier |
| Moscow      |
| New York    |
| Paris       |
| Tokyo       |
---------------
```

▼ `FROM` clause

Specifies which table we want the query to return data from e.g.,

```
SELECT column1, column2, column3
FROM table1
```

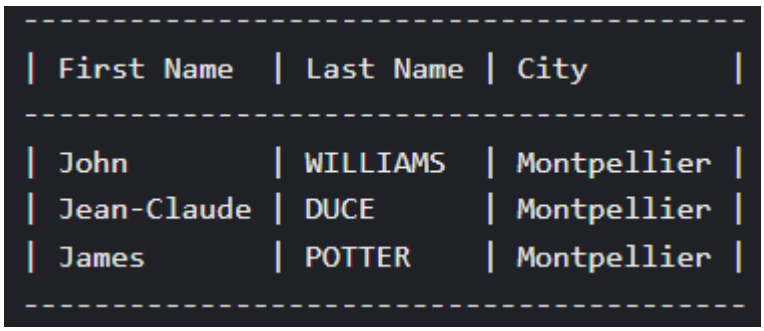
▼ Filters: `WHERE` clause

- Used to retrieve specific rows from tables
- Requires a **conditional statement**

The query

```
SELECT firstname AS "First Name", lastname AS "Last Name", city AS "City"
FROM customer
WHERE city = 'Montpellier';
```

returns



First Name	Last Name	City
John	WILLIAMS	Montpellier
Jean-Claude	DUCE	Montpellier
James	POTTER	Montpellier

▼ Comparison operators

`<`	Less than
`>`	More than
`<=`	Less than or equal to
`>=`	More than or equal to
`!=`	Not equal to

```
SELECT *
FROM product
WHERE available_stock < 100;
```

```
SELECT *
FROM customer
WHERE city = 'New York';
```

Multiple Comparisons

Use `AND` and `OR` to chain filters

```
SELECT *
FROM product
WHERE price < 20 AND available_stock > 1000;
```

```
SELECT *
FROM customer
WHERE lastname = 'White' OR lastname = 'Williams'
```

▼ Wildcards

- Serve as substitutes for other characters
- Used in conjunction with the `LIKE` keyword in a `WHERE` clause

▼ Underscore `_`

- Substitutes a single character

```
SELECT *
FROM customer
WHERE firstname LIKE 'Li_a'; -- Would return 'Lisa', 'Lita', 'Liza' etc.
```

▼ Percentage `%`

- Substitutes for 0 or more characters

```
SELECT *
FROM customer
WHERE firstname LIKE 'J%' -- returns all names beginning with J
```

▼ `[ABC]`

- Specify multiple possible characters to match

```
SELECT *
FROM product
WHERE name LIKE 'Kindle Fire [56]' -- returns e.g., 'Kindle Fire 5', 'Kindl
e Fire 6'
```

▼ `[^ABC]`

- The circumflex character is used to *negate* characters inside the square brackets

```
SELECT *
FROM product
WHERE name LIKE 'Kindle Fire [^1234]' -- will not return e.g., 'Kindle Fire
1'.
```

▼ Combining wildcards

```
SELECT *
FROM product
WHERE name LIKE '[ABC]%' -- returns all products starting with 'A', 'B' and
'C'
```

```
SELECT *
FROM product
WHERE name LIKE '&:%' -- -- returns all products with a : in the name
```

```
SELECT name, price, description
FROM product
WHERE description LIKE '%HD%'; -- returns all products where description in
cludes 'HD'
```

```
| NAME              | PRICE  | DESCRIPTION                                  |
| Kindle Fire 7     | 119.00 | Kindle Fire 7 Tablet, Now in HD              |
| Chromecast        | 35.00  | Google Chromecast HDMI Streaming Media Player |
| Samsung UE46F6100 | 589.99 | Samsung UE46F6100 TV LCD 46 (117 cm) LED 3D HD |
```

▼ `BETWEEN` keyword

Can be used in place of comparison operators i.e.,

```
SELECT *
FROM product
WHERE price BETWEEN 50 AND 100;
```

is the same as:

```
SELECT *
FROM product
WHERE price >= 50 AND price <= 100;
```

Can also be chained with other filters:

```
SELECT name, description, price, available_stock
FROM product
where available_stock BETWEEN 10 AND 20 AND price > 100;
```

```
| NAME            | DESCRIPTION                          | PRICE  | AVAILABLE_STOCK |
| GreenWorks 25022 | 12 Amp 20-in 3-in-1 Electric Lawn Mower | 172.73 | 20              |
| VIFAH V501      | Outdoor Wood Serving Cart            | 251.49 | 12              |
| ProForm 6.0 RT  | Treadmills                           | 499.99 | 12              |
```

▼ `IN` keyword

- Use in place of chained ORs i.e.,

```
SELECT *
FROM product
WHERE firstname IN ('James', 'Roger', 'Jean-Claude');
```

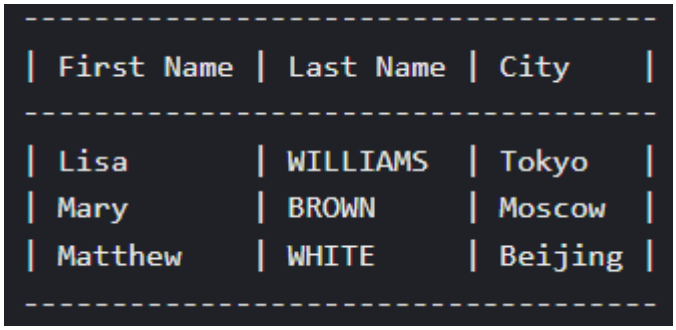
is the same as:

```
SELECT *
FROM product
WHERE firstname = 'James' OR firstname = 'Roger' OR firstname = 'Jean-Claude');
```

The following query

```
SELECT firstname AS "First Name", lastname AS "Last Name", city AS "City"
FROM customer
WHERE city IN ('Tokyo', 'Beijing', 'Moscow');
```

will return



First Name	Last Name	City
Lisa	WILLIAMS	Tokyo
Mary	BROWN	Moscow
Matthew	WHITE	Beijing

▼ `NULL`

- Is a non-value i.e.,
- Is not equal to zero
- Is not equal to an empty string
- Is not equal to anything…
- not even `NULL`

Using,

```
WHERE birth_date = NULL
```

to find records in which the `birth_date` is missing will NOT work, because nothing is equal to `NULL`.

Instead, we must use the `IS` keyword:

```
SELECT *
FROM customer
WHERE birth_date IS NULL;
```

To find records without missing data, we can use `IS NOT NULL`:

```
SELECT *
FROM customer
WHERE birth_date IS NOT NULL;
```

Exercise

The customer table contains the following columns:

- `customer_id`
- `firstname`
- `lastname`
- `register_date`
- `birth_date`
- `address`
- `zipcode`
- `city`
- `phone_number`

Return only the `firstname`, `lastname`, `address` and `phone-number` columns from the customer table, but give them the column headings “First Name”, “Last Name”, “Address” and “Phone Number”. Only return rows for customers that are missing either an address or a phone number.

```
SELECT firstname AS "First Name", lastname AS "Last Name", address AS "Add
ress", phone_number AS "Phone Number"
FROM customer
WHERE address IS NULL OR phone_number IS NULL;
```

First Name	Last Name	Address	Phone Number
Lisa	WILLIAMS	null	03-6487-3260
Mary	BROWN	null	null
Paul	HARRIS	null	(718) 555-5987
Elizabeth	JACKSON	23, Istiklal Avenue	null
Matthew	WHITE	null	null
Jean-Claude	DUCE	null	null
James	POTTER	null	null
Roger	POULTON	null	null

▼ `TOP` keyword

So far, we've returned specific columns with `SELECT` and specific rows with `WHERE`.

We can also filter rows by using the `TOP` keyword:

```
SELECT TOP 5
FROM customer;
```

▼ `GROUP BY` clause

- Used to group data together
- Almost always used in conjunction with aggregate functions
- Everything in the `SELECT` clause **must** be an aggregate, or appear in the `GROUP BY` clause

▼ Aggregation

- The act of bringing things together into groups
- We can return multiple aggregate functions by a single query

```
SELECT SUM(price) AS "Total Price"
FROM product;
```

```
SELECT SUM(available_stock) AS "Total Stock",
        AVG(available_stock) AS "Average Stock",
        MIN(available_stock) AS "Minimum Stock",
        MAX(available_stock) AS "Maximum Stock",
        COUNT(available_stock) AS "Number of products with Non-Null Stock"
FROM product;
```

```
SELECT SUM(available_stock * price) AS "Total Value"
FROM product;
```

The following query calculates the average of the price column for each `product_category_id`, and counts the number of names there are in the `product_category` table.

```
SELECT product_category_id AS "Category ID",
  AVG(price) AS "Average Price", -- aggregated data
  COUNT(name) AS "Number of Products" -- aggregated data
FROM product
GROUP BY product_category_id;
```

▼ Exercise: `GROUP BY`

The customer table contains the following columns:

- `customer_id`
- `firstname`
- `lastname`
- `register_date`

- `birth_date`
- `address`
- `zipcode`
- `city`
- `phone_number`

Return the cities in the customer table along with the number of customers from each city. Give your columns the aliases `“City”` and `“Customer Count”`. Order your results by the count, with the largest count at the top, and in alphabetical order of city to decide any ties.

```
SELECT city AS "City",
    COUNT(customer_id) AS "Customer Count"
FROM customer
GROUP BY city
ORDER BY COUNT(customer_id) DESC, city ASC;
```

▼ `HAVING` clause

- Use the HAVING keyword if you want to filter the result of an aggregation

The following SQL query is used to retrieve the average price of each product category that has an average price less than 200.

```
SELECT product_category_id AS "Category ID",
  AVG(price) AS "Average Price" -- SELECT is processed fourth
FROM product -- FROM is processed first
GROUP BY product_category_id -- GROUP BY is processed second
HAVING AVG(price) < 200; -- HAVING is processed third
```

Here's a breakdown of each step:

1. `SELECT product_category_id AS "Category ID", AVG(price) AS "Average Price"`: This line specifies the columns to be selected. The `SELECT` statement retrieves two columns, namely `product_category_id` and `AVG(price)`. The alias `Category ID`

and `Average Price` are assigned to each column, respectively. The `AVG()` function computes the average price of each product category.

2. `FROM product`: This line specifies the table from which to select the columns. The table `product` is selected in this case.
3. `GROUP BY product_category_id`: This line groups the records in the `product` table by `product_category_id`. This is used in combination with the `AVG()` function to compute the average price of each product category.
4. `HAVING AVG(price) < 200`: This line is used to filter the records based on a condition. In this case, only records with an average price less than 200 are selected. The `HAVING` keyword is used to filter records based on the result of the aggregate function `AVG()`, whereas the `WHERE` keyword is used to filter records based on a single row.

In summary, the query retrieves the average price of each product category and filters out the categories whose average price is greater than or equal to 200. The output includes two columns, `Category ID` and `Average Price`.

⚠️ Note that we have to recalculate the aggregate - we could not refer to AVG(price) using the “Average Price” alias.

⚠️ Do not confuse HAVING and WHERE. WHERE is used to filter rows in the original table; HAVING is used to filter the aggregated data. Both can be used together.

```
SELECT product_category_id AS "Category ID",
  AVG(price) AS "Average Price
FROM product
WHERE product_category_id IS NOT NULL
GROUP BY product_category_id
HAVING AVG(price) < 200;
```

Exercise

The product table contains the following columns:

- `product_id`
- `product_category_id`
- `name`
- `description`
- `price`
- `available_stock`

Return a list of the product category IDs and the highest product price for each category. Give your columns the aliases `“Category ID”` and `“Maximum Price”`. Only return results when the minimum available stock for products in that category is less than 100. Order your results by Category ID (ascending).

```
SELECT product_category_id AS "Category ID",
  MAX(price) AS "Maximum Price"
FROM product
WHERE product_category_id IS NOT NULL
GROUP BY product_category_id
HAVING MIN(available_stock) < 100
ORDER BY product_category_id ASC;
```

▼ `ORDER BY` clause

- the last clause in the Syntax Sequence
- *and* the last clause in the Processing Sequence
- sorts results based on the columns (or calculations) specified
- Text can be ordered by ascending order through use of the `ASC` keyword, and by descending order through use of the `DESC` keyword. (if we don’t specify it will be ordered by ascending order by default)

```
SELECT *
FROM customer
ORDER BY lastname ASC, firstname ASC;
```

▼ Exercise: `ORDER BY` clause

The product table contains the following columns:

- `product_id`
- `product_category_id`
- `name`
- `description`
- `price`
- `available_stock`

Return only the name, description, and price columns. Only return rows for the five most expensive products.

```
SELECT TOP 5 price, name, description,
FROM product
ORDER BY price DESC;
```

```
-------------------------------------------------------------------------------------------------
| PRICE  | NAME                                  | DESCRIPTION                                  |
-------------------------------------------------------------------------------------------------
| 999.99 | Intex Rectangular Ultra Frame Pool Set | 24-Feet by 12-Feet by 52-Inch                |
| 589.99 | Samsung UE46F6100                     | Samsung UE46F6100 TV LCD 46 (117 cm) LED 3D HD |
| 499.99 | ProForm 6.0 RT                        | Treadmills                                   |
| 301.22 | DuroStar DS4000S                      | 4-Cycle Gas Powered Portable Generator       |
| 299.00 | Weber 781001                          | Gold One-Touch 26-1/2-Inch Kettle Grill, Black |
-------------------------------------------------------------------------------------------------
```

▼ Data Manipulation

▼ Concatenation

- We can combine text from multiple columns by means of concatenation.

> ⚠️ Different RDBMSs may use slightly different syntax for concatenation

For Microsoft SQL Server, we use `+` symbols to combine columns:

```
SELECT firstname + ' ' + lastname AS "Full Name"
FROM customer;
```

For ANSI SQL (used by CodinGame), replace the `+` symbols with `||` symbols:

```
SELECT firstname || ' ' || lastname AS "Full Name"
FROM customer
```

⚠️ ALWAYS use aliases when combining columns, else the column header will be `'No column name'`.

▼ Exercise: Concatenation

The customer table contains the following columns:

- `customer_id`
- `firstname`
- `lastname`
- `register_date`
- `birth_date`
- `address`
- `zipcode`
- `city`
- `phone_number`

Concatenate the address and city fields with a comma and space between them e.g., “123 Sparta Street, London”. Give this column the alias “Full

Address”. Do not include results for any customers without an address or city listed.

```
SELECT address + ', ' + city AS "Full Address"
FROM customer
WHERE address IS NOT NULL AND city IS NOT NULL;
```

```
----------------------------------------
| Full Address                         |
----------------------------------------
| 16 rue de l'eglise, Paris            |
| 144 avenue de l'europe, Montpellier  |
| 23, Istiklal Avenue, Istanbul        |
----------------------------------------
```

▼ Arithmetic

We can manipulate numerical data through means of arithmetic operators:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulo

```
SELECT name AS "Name"
  available_stock AS "Available Stock"
  available_stock % 10 "Remaining Stock"
FROM product;
```

▼ Exercise: Arithmetic

The product table contains the following columns:

- `product_id`
- `product_category_id`
- `name`
- `description`
- `price`
- `available_stock`

Return only the name, price and `available_stock` columns from the product table along with a calculated column showing the total value of all available stock. Give your columns the aliases “Name”, “Price”, “Available Stock” and “Total Value”.

```
SELECT name AS "Name", price AS "Price", available_stock AS "Available Stoc
k",
    price * available_stock AS "Total Value"
FROM product;
```

Name	Price	Available Stock	Total Value
Kindle Fire 7	119.00	567	67473.00
Leap motion	94.00	864	81216.00
Chromecast	35.00	65	2275.00
Samsung UE46F6100	589.99	21	12389.79
TomTom XL Classic Europe 23	90.20	15	1353.00
The Orphan Master's Son: A Novel	9.57	53	507.21
War of the Whales: A True Story	18.34	3287	60283.58
The Fracking King: A Novel	13.80	150	2070.00
California: A Novel	18.09	1237	22377.33
DuroStar DS4000S	301.22	3287	990110.14
Weber 781001	299.00	58	17342.00
Intex Rectangular Ultra Frame Pool Set	999.99	7	6999.93
GreenWorks 25022	172.73	20	3454.60
VIFAH V501	251.49	12	3017.88
ProForm 6.0 RT	499.99	12	5999.88
Weider 190 RX	100.90	28	2825.20
Ab Wheel	17.99	98	1763.02
ACDelco CF178	23.13	12	277.56
Wilwood 260-11179	67.99	89	6051.11
Electrical PVC Insulation Adhesive Tape	2.68	654	1752.72
HC-SR04	2.71	145	392.95

▼ `CASE` keyword

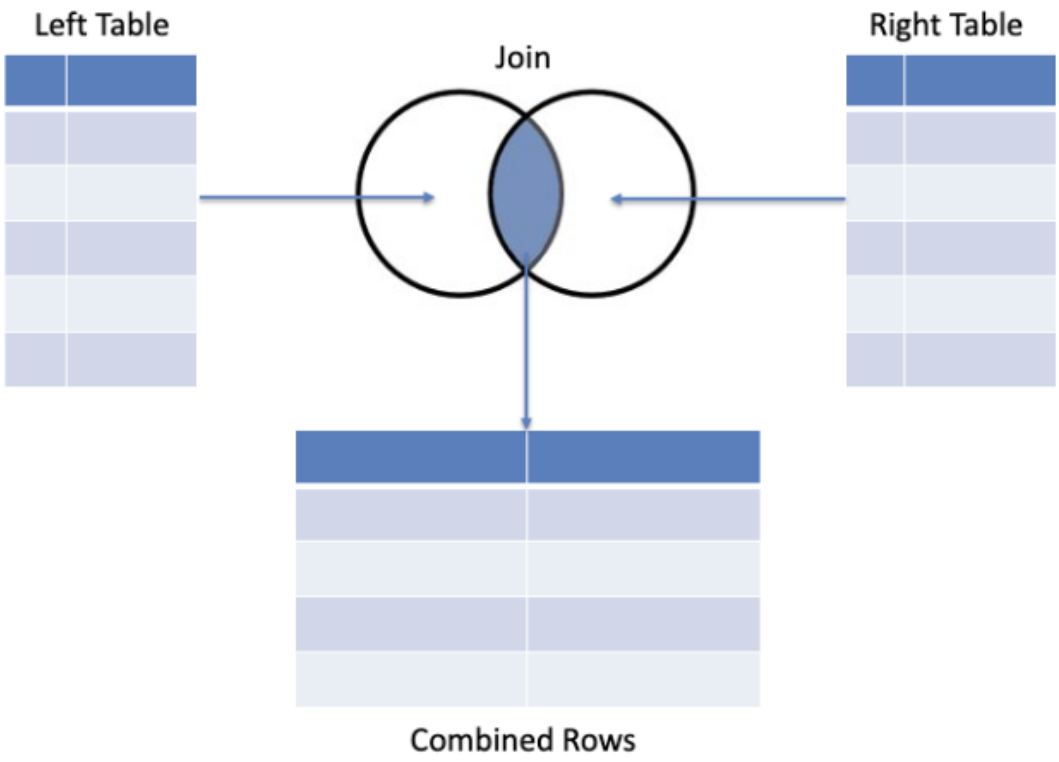
- We begin a `CASE` statement by means of the `CASE` keyword
- We specify each condition by means of the `WHEN` keyword, and that condition's outcome by means of the `THEN` keyword
- Any conditions not specified by `WHEN` are described by `ELSE`
- Conditions are executed line-by-line
- We terminate a `CASE` statement by means of the `END` keyword, and give the column an alias

```
SELECT name, price
  CASE
    WHEN price < 50 THEN 'Cheap'
    WHEN price < 100 THEN 'Moderately Priced'
    ELSE 'Expensive'
  END AS "Price Category"
FROM product;
```

```
SELECT name, description, price, available_stock,
    CASE
        WHEN available_stock < 20 THEN 'Low Stock!'
        WHEN available_stock < 100 THEN 'Limited Stock'
        ELSE 'Well-Stocked'
    END AS "STOCK LEVEL"
FROM product;
```

▼ `JOIN` keyword

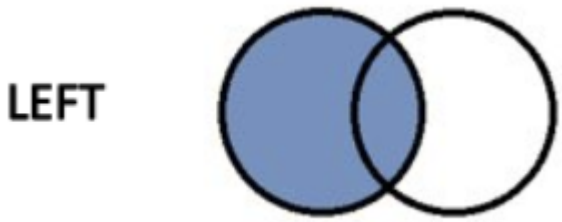
- Combine tables by using the `JOIN` keyword
- Use as part of a `FROM` clause



▼ Types of `JOIN`

`LEFT JOIN`

A `LEFT JOIN` (or `LEFT OUTER JOIN`) returns all rows in the Left Table, and only returns rows from the Right Table that match with rows in the Left Table.

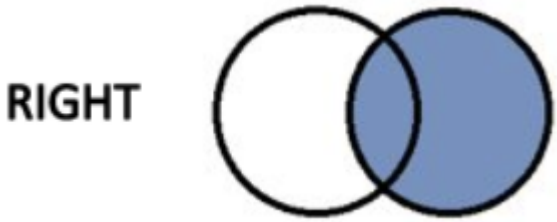



```
SELECT *
FROM LeftTable
LEFT JOIN RightTable
ON LeftTable.KeyColumn = RightTable.KeyColumn
```

`RIGHT JOIN`

A `RIGHT JOIN` (or `RIGHT OUTER JOIN`) returns all rows in the Right Table, and only returns rows from the Left Table that match with rows in the Right Table.

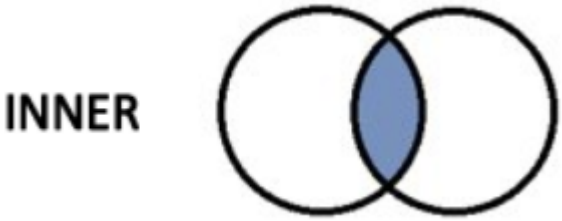
> ⚠️ RIGHT JOINS can always be refactored as LEFT JOINS and should generally be avoided.




```
SELECT *
FROM LeftTable
RIGHT JOIN RightTable
ON LeftTable.KeyColumn = RightTable.KeyColumn
```

`INNER JOIN`

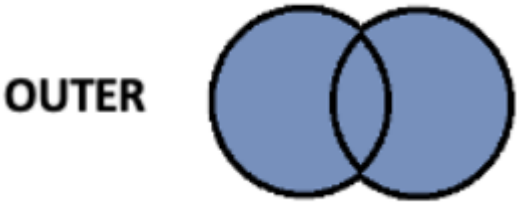
An `INNER JOIN` (or simply `JOIN`) returns rows from each table ONLY if there is a match in the other table.




```
SELECT *
FROM LeftTable
INNER JOIN RightTable
ON LeftTable.KeyColum = RightTable.KeyColumn
```

`OUTER JOIN`

An `OUTER JOIN` (or `FULL OUTER JOIN)` returns all rows from each table, regardless of whether there is a match with the other table.

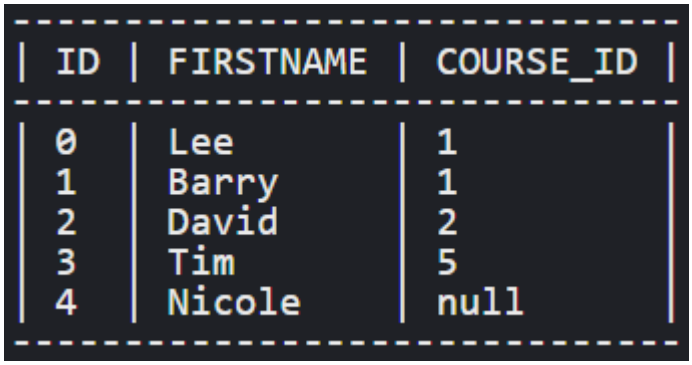



```
SELECT *
FROM LeftTable
FULL OUTER JOIN RightTable
ON LeftTable.KeyColumn = RightTable.KeyColumn
```

⚠️ For `LEFT`, `RIGHT`, and `OUTER JOIN`s, whenever there is not a matching row in one of the tables, the missing values will be represented by `NULL`.

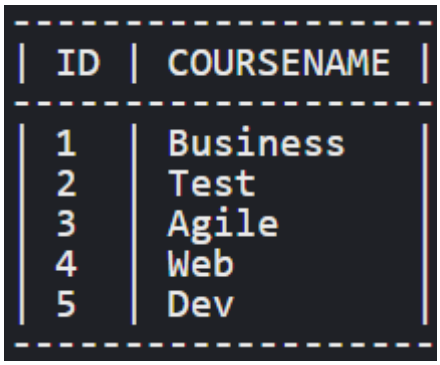
Exercise

The contents of a table, student, are shown below:



ID	FIRSTNAME	COURSE_ID
0	Lee	1
1	Barry	1
2	David	2
3	Tim	5
4	Nicole	null

The contents of a table, course, are shown below:



ID	COURSENAME
1	Business
2	Test
3	Agile
4	Web
5	Dev

Write a query that joins the two tables. Match the `course_id` column in the students table with the `id` column in the course table. Choose the type of join so that only rows that have a match in both tables are returned.

```
SELECT student.firstname, course.coursename
FROM student
-- ADD YOUR JOIN STATEMENT HERE
-- To join only matching rows, we should use an INNER JOIN
INNER JOIN course
ON student.course_id = course.id
```

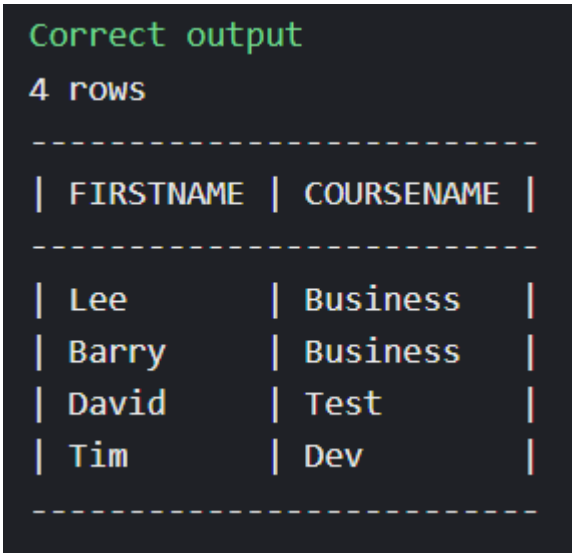


Table Aliases

Instead of using the `AS` keyword to give our tables aliases, we can save time and effort by using dot notation

Standard practice is to use the initials of the tables, adding letters if there are conflicting names.

```
SELECT s.firstname, c.coursename
FROM student s
INNER JOIN course c
ON s.course_id = c.id
```

Using column aliases and table aliases:

```
SELECT s.firstname AS "First Name", c.coursename AS "Course"
FROM student s
LEFT JOIN course c
ON s.course_id = c.id
```

▼ Multiple `JOIN`s

We can perform multiple joins with a single query:

```
SELECT *
FROM Table_1 t1
JOIN Table_2 t2
  ON t1.id = t2.id
JOIN Table_3 t3
  ON t1.id = t3.id
```

Each new `JOIN` joins on to the original table specified at the start of the `FROM` clause, or any table that has already been joined on to this table.

In the following example, Table_3 also has a column (attribute) in common with Table_1, so that is the basis of the `JOIN`.

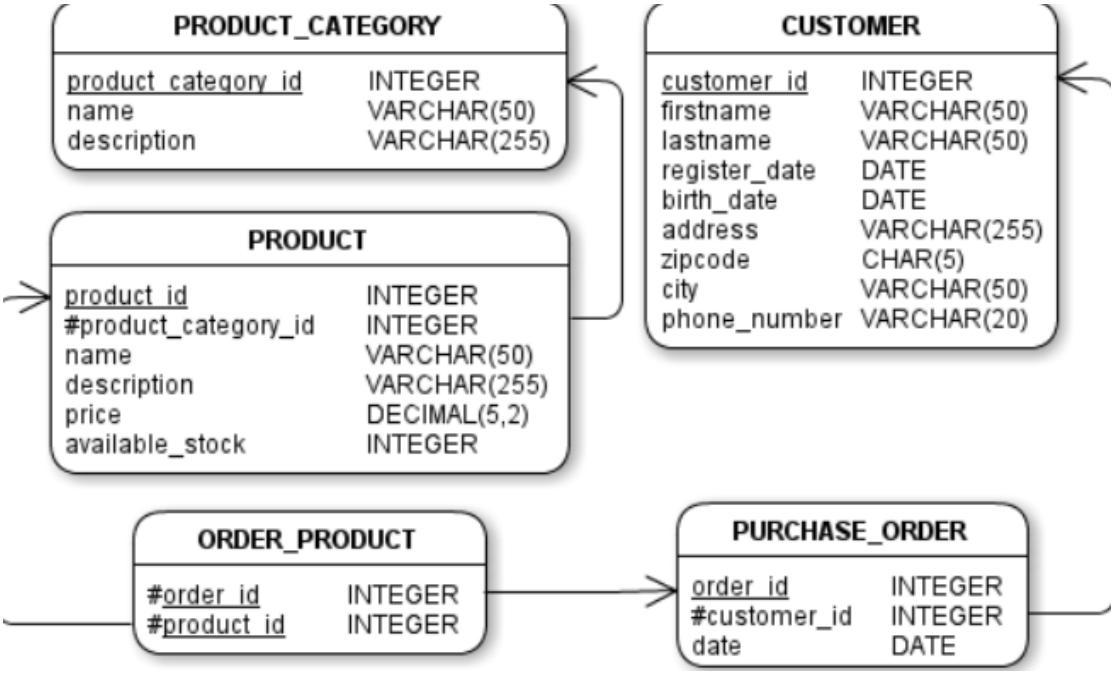
```
SELECT *
FROM Table_1 t1
JOIN Table_2 t2
  ON t1.id = t2.id
JOIN Table_3 t3
  ON t1.id = t3.id
```

In the following example, Table_3 also has a column (attribute) in common with Table_2, so that is the basis of the `JOIN`.

```
SELECT *
FROM Table_1 t1
JOIN Table_2 t2
  ON t1.id = t2.id
JOIN Table_3 t3
  ON t2.id = t3.id
```

Exercise

A database design is outlined in an ERD below:



Using multiple `JOIN`s, produce the following columns.

- Purchase Order Date (”Date”)
- The First Name and Last Name of the customer making the purchase, in a single column, in upper case, separated by a space e.g., JOHN SMITH (”Customer”)
- The name of the product purchased (”Product”)
- The name of the category of the purchased product (”Product Category”)

Assign them the alias shown in parenthesis (). Order by date, with the oldest purchases shown first.

Tips

- Study the ERD
- Focus on the `JOIN`s first, then on the `SELECT` clause
- This will require joining every single table in the database, even if you don’t `SELECT` columns from every table

```
SELECT po.date AS "Date",
  UPPER(c.firstname || ' ' || c.lastname) AS "Customer",
  p.name AS "Product", pc.name AS "Product Category"
FROM product_category pc
JOIN product p
  ON pc.product_category_id = p.product_category_id
JOIN order_product op
```

```
  ON p.product_id = op.product_id
JOIN purchase_order po
  ON op.order_id = po.order_id
JOIN customer c
    ON po.customer_id = c.customer_id
ORDER BY po.date ASC;
```

Date	Customer	Product	Product Category
1999-12-29	PAUL HARRIS	Chromecast	High-tech
1999-12-29	PAUL HARRIS	Wilwood 260-11179	Automotive
2002-04-25	PAUL HARRIS	California: A Novel	Books
2002-04-25	PAUL HARRIS	ProForm 6.0 RT	Fitness
2002-04-25	PAUL HARRIS	Weider 190 RX	Fitness
2002-04-25	PAUL HARRIS	ACDelco CF178	Automotive
2002-04-25	PAUL HARRIS	Wilwood 260-11179	Automotive
2003-04-08	PAUL HARRIS	Weber 781001	Garden
2003-04-08	PAUL HARRIS	VIFAH V501	Garden
2003-04-14	MARY BROWN	ProForm 6.0 RT	Fitness
2005-08-14	PAUL HARRIS	Intex Rectangular Ultra Frame Pool Set	Garden
2005-08-14	PAUL HARRIS	ACDelco CF178	Automotive
2009-01-03	ELIZABETH JACKSON	Leap motion	High-tech
2009-01-03	ELIZABETH JACKSON	Wilwood 260-11179	Automotive
2010-12-29	LISA WILLIAMS	Ab Wheel	Fitness
2011-11-01	ELIZABETH JACKSON	The Fracking King: A Novel	Books
2011-12-29	LISA WILLIAMS	Leap motion	High-tech
2011-12-29	LISA WILLIAMS	California: A Novel	Books
2012-10-03	JAMES SMITH	Kindle Fire 7	High-tech
2012-10-03	JAMES SMITH	TomTom XL Classic Europe 23	High-tech
2014-01-07	LISA WILLIAMS	The Orphan Master's Son: A Novel	Books
2014-03-05	JAMES SMITH	DuroStar DS4000S	Garden
2014-03-05	JAMES SMITH	GreenWorks 25022	Garden