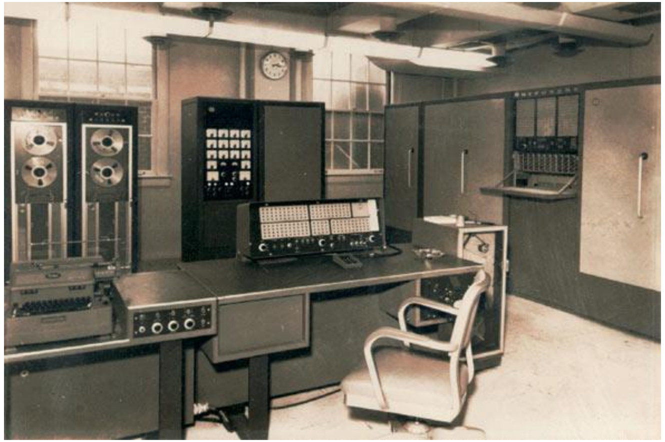
Introduction to Structured Databases II

THE KNOWLEDGE HOUSE

Agenda - Schedule

1. SQL Leetcode Q
2. SQL Review
3. CASE Statements & Subqueries
4. Normalization
5. Break
6. SQL Murder Mystery



Database systems of the past

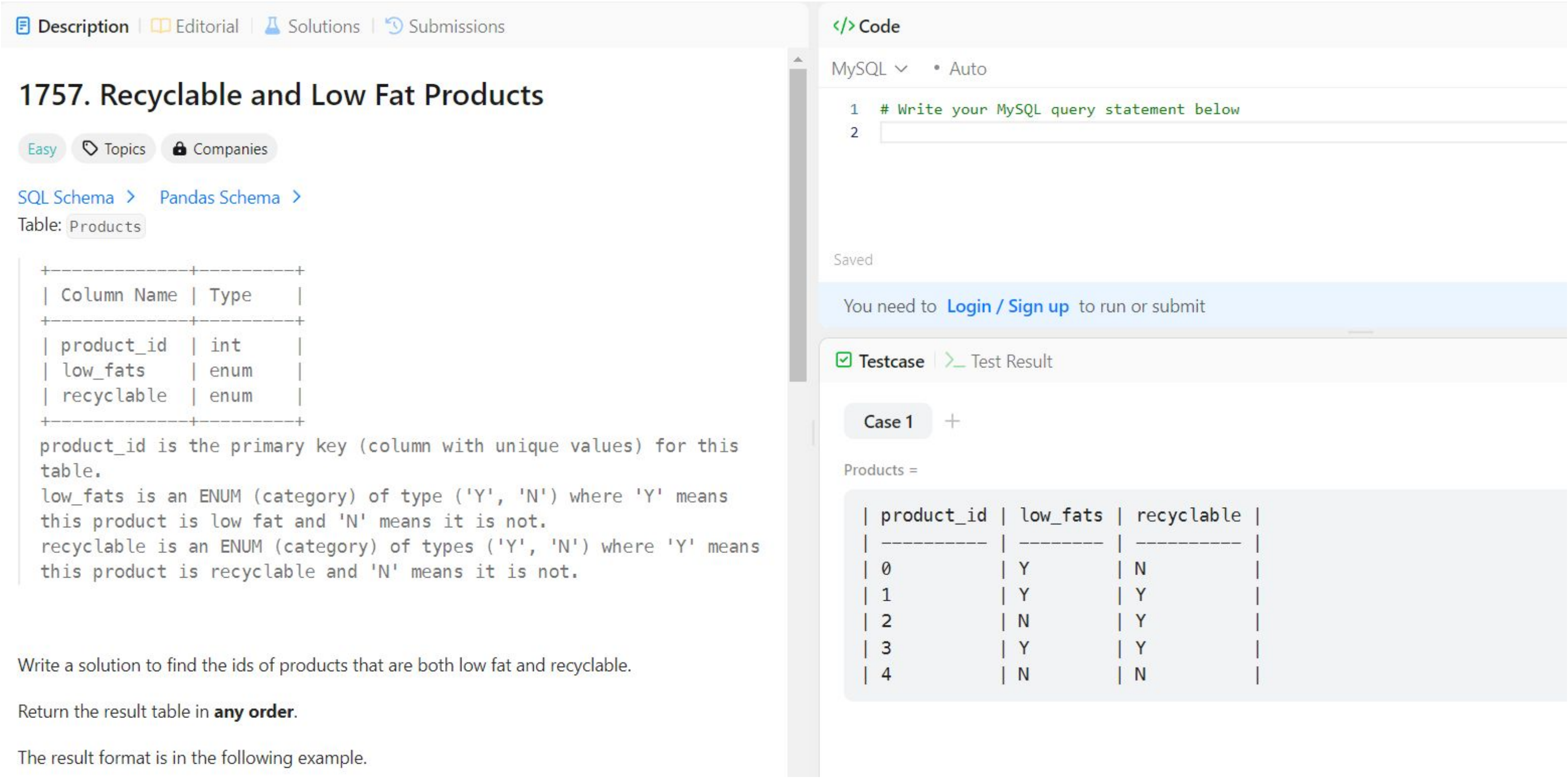
Agenda - Announcements

- Week 9 Pre-Class Quiz due 5/9 *(2 attempts)*
- TLAB #3 due 5/14
  - Early grade due date: **5/7**
  - Extension due date: **5/13**

Agenda - Goals

- Write subqueries to filter, summarize, or feed into larger SQL statements
- Use CASE statements to apply conditional logic in SELECT queries
- Understand, at a high level, why database normalization helps improve clarity and reduce redundancy
- Apply all your skills from this week to solve a real-world-style data challenge using SQL

SQL Leetcode Q



1757. Recyclable and Low Fat Products

Table: `Products`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| low_fats    | enum    |
| recyclable  | enum    |
+-------------+---------+
product_id is the primary key (column with unique values) for this
table.
low_fats is an ENUM (category) of type ('Y', 'N') where 'Y' means
this product is low fat and 'N' means it is not.
recyclable is an ENUM (category) of types ('Y', 'N') where 'Y' means
this product is recyclable and 'N' means it is not.
```

Write a solution to find the ids of products that are both low fat and recyclable.

Return the result table in **any order**.

The result format is in the following example.

```
# Write your MySQL query statement below

```

You need to **Login / Sign up** to run or submit

Products =

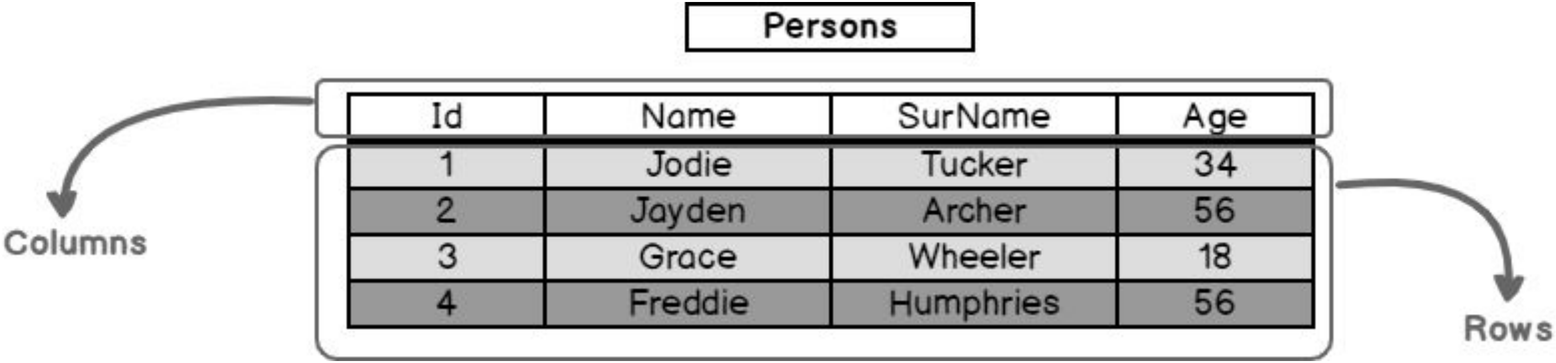
```
| product_id | low_fats | recyclable |
| ---------- | -------- | ---------- |
| 0          | Y        | N          |
| 1          | Y        | Y          |
| 2          | N        | Y          |
| 3          | Y        | Y          |
| 4          | N        | N          |
```

Take some time to complete Recyclable & Low Fat Products, **this time IN SQL**
https://leetcode.com/problems/recyclable-and-low-fat-products/description/?envType=study-plan-v2&envId=top-sql-50

SQL Overview/Review

Let’s go back to basics for a moment and discuss SQL’s structure

SQL databases have a bunch of tables, these tables have columns and rows



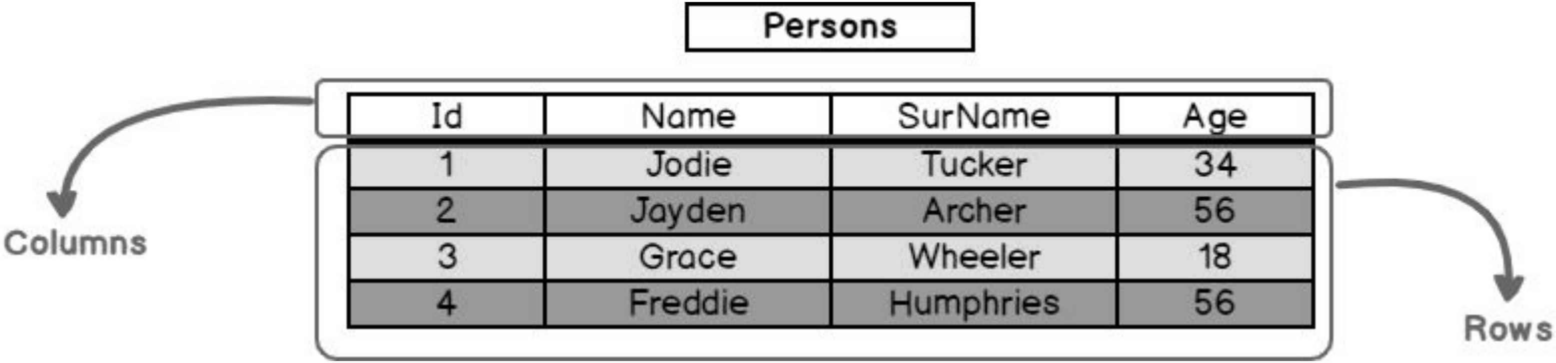
Persons

Id	Name	SurName	Age
1	Jodie	Tucker	34
2	Jayden	Archer	56
3	Grace	Wheeler	18
4	Freddie	Humphries	56

SQL Review

SQL Overview/Review

Each column can have different types, here are some of the main ones:
INTEGER
VARCHAR(length) - this is equivalent to a Python string but we can set a maximum length
FLOAT
BOOL



Persons

Id	Name	SurName	Age
1	Jodie	Tucker	34
2	Jayden	Archer	56
3	Grace	Wheeler	18
4	Freddie	Humphries	56

SQL Overview/Review

In order to get data from our table we MUST

SELECT *columns*
FROM *table*

When it is just *one* table, we don't need to specify the name

When we do **joins**, we need to specify the name if the columns names are shared

However, **every table must be explicitly named**

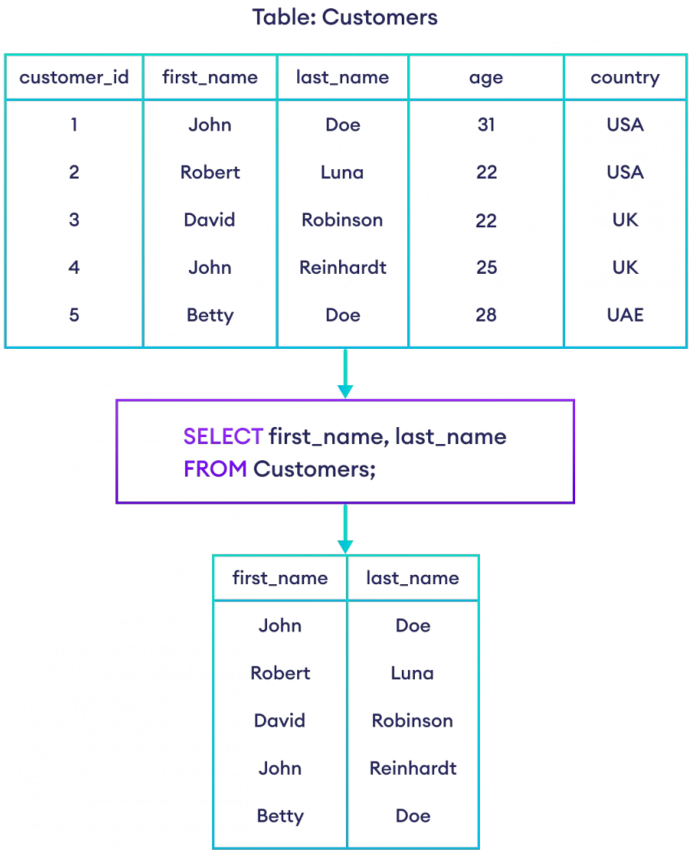


Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT first_name, last_name
FROM Customers;
```

first_name	last_name
John	Doe
Robert	Luna
David	Robinson
John	Reinhardt
Betty	Doe

SQL Overview/Review

We can also alias columns **and** tables, so on the right we could o

```
SELECT first_name AS first,
       last_name AS last
FROM Customers AS c
```

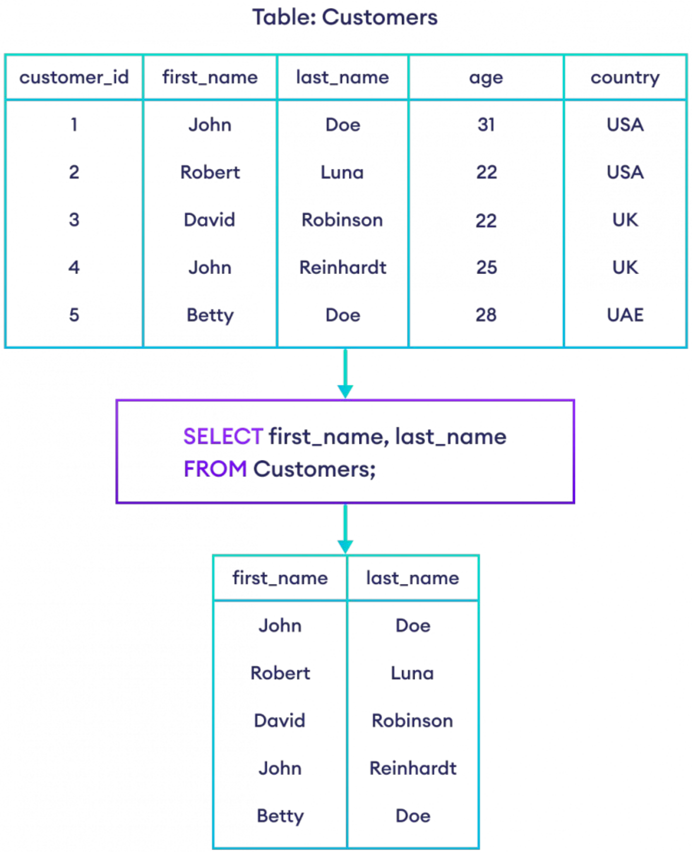


Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT first_name, last_name
FROM Customers;
```

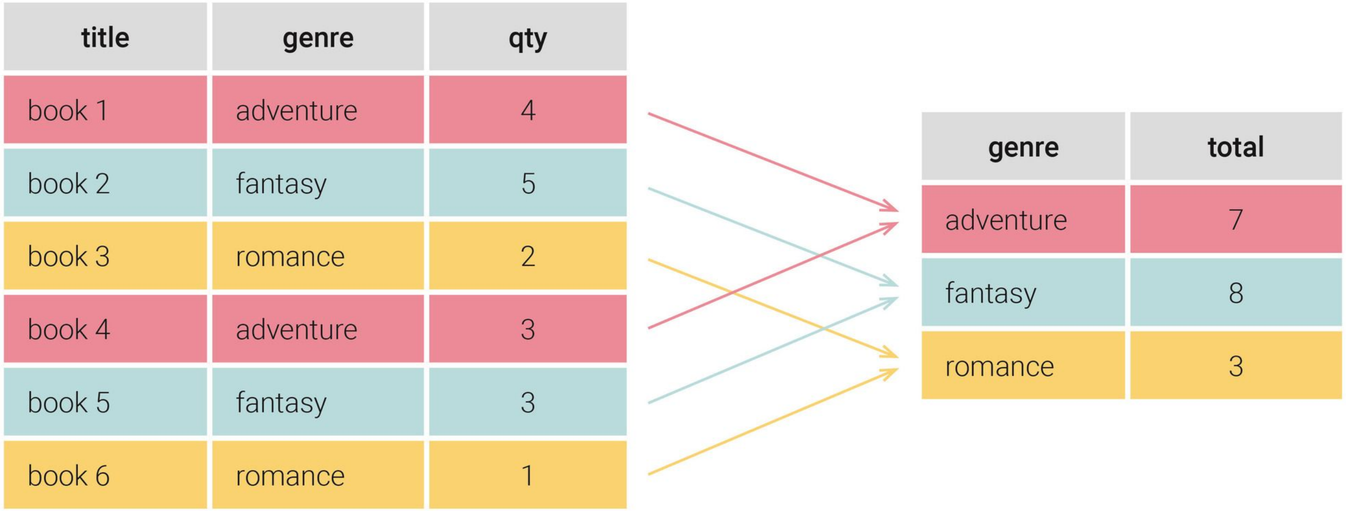
first_name	last_name
John	Doe
Robert	Luna
David	Robinson
John	Reinhardt
Betty	Doe

SQL GROUP BY

We use GROUP BY to combine (group) on column variables to get a result
The below query could be

```
SELECT genre,
SUM(qty)
FROM books
GROUP BY genre
```

Note how the genre column “collapses” into the unique values and that is added together

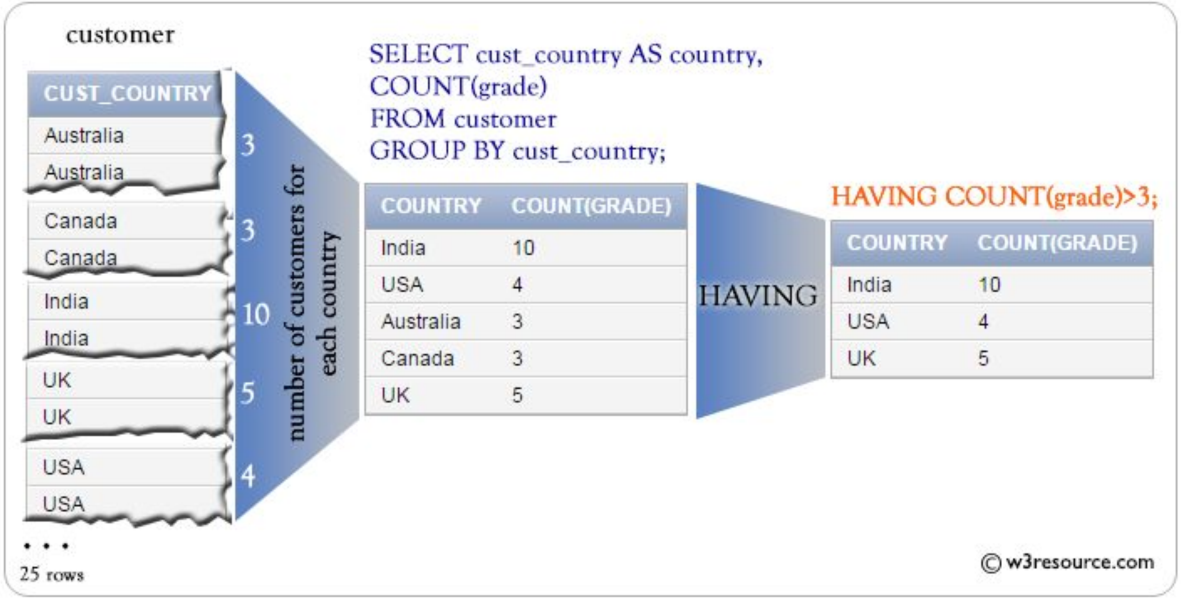


title	genre	qty
book 1	adventure	4
book 2	fantasy	5
book 3	romance	2
book 4	adventure	3
book 5	fantasy	3
book 6	romance	1

genre	total
adventure	7
fantasy	8
romance	3

SQL Filtering

HAVING must be used with a **GROUP BY** statement, if we try to use **HAVING** without **GROUP BY** we will get an error



customer

CUST_COUNTRY
Australia
Australia
Canada
Canada
India
India
UK
UK
USA
USA

• • •
25 rows

```
SELECT cust_country AS country,
COUNT(grade)
FROM customer
GROUP BY cust_country;
```

COUNTRY	COUNT(GRADE)
India	10
USA	4
Australia	3
Canada	3
UK	5

```
HAVING COUNT(grade)>3;
```

COUNTRY	COUNT(GRADE)
India	10
USA	4
UK	5

© w3resource.com

SQL JOINS

Combining Data Tables – SQL Joins Explained

A JOIN clause in SQL is used to combine rows from two or more tables, based on a related column between them.

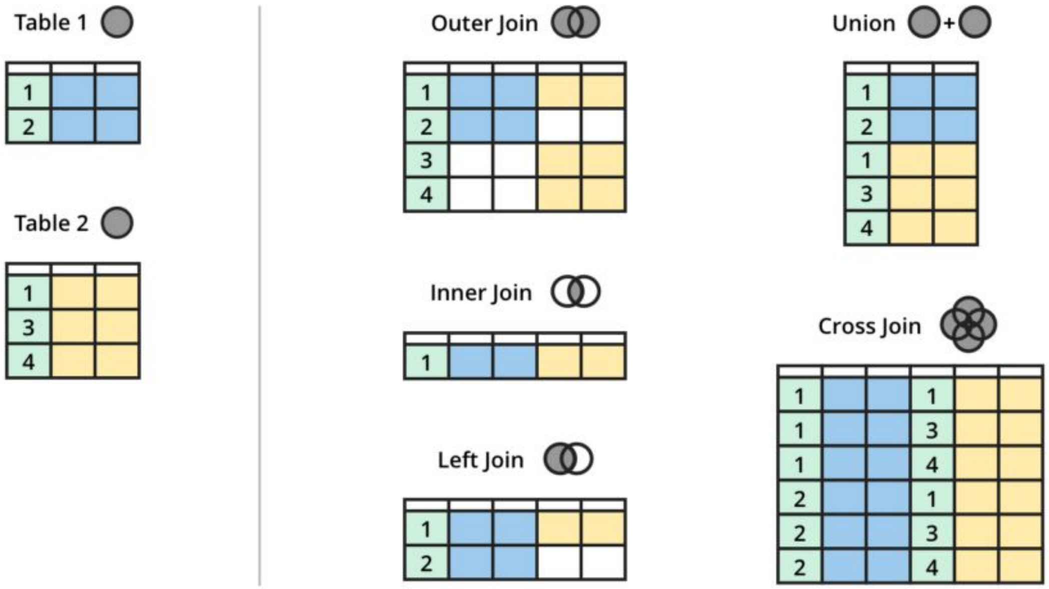


Table 1 ●

1	🟦	🟦
2	🟦	🟦

Table 2 ●

1	🟨	🟨
3	🟨	🟨
4	🟨	🟨

Outer Join

1	🟦	🟦	🟨	🟨
2	🟦	🟦		
3			🟨	🟨
4			🟨	🟨

Inner Join

1	🟦	🟦	🟨	🟨

Left Join

1	🟦	🟦	🟨	🟨
2	🟦	🟦		

Union ● + ●

1	🟦	🟦
2	🟦	🟦
1	🟨	🟨
3	🟨	🟨
4	🟨	🟨

Cross Join

1	🟦	🟦	1	🟨	🟨
1	🟦	🟦	3	🟨	🟨
1	🟦	🟦	4	🟨	🟨
2	🟦	🟦	1	🟨	🟨
2	🟦	🟦	3	🟨	🟨
2	🟦	🟦	4	🟨	🟨

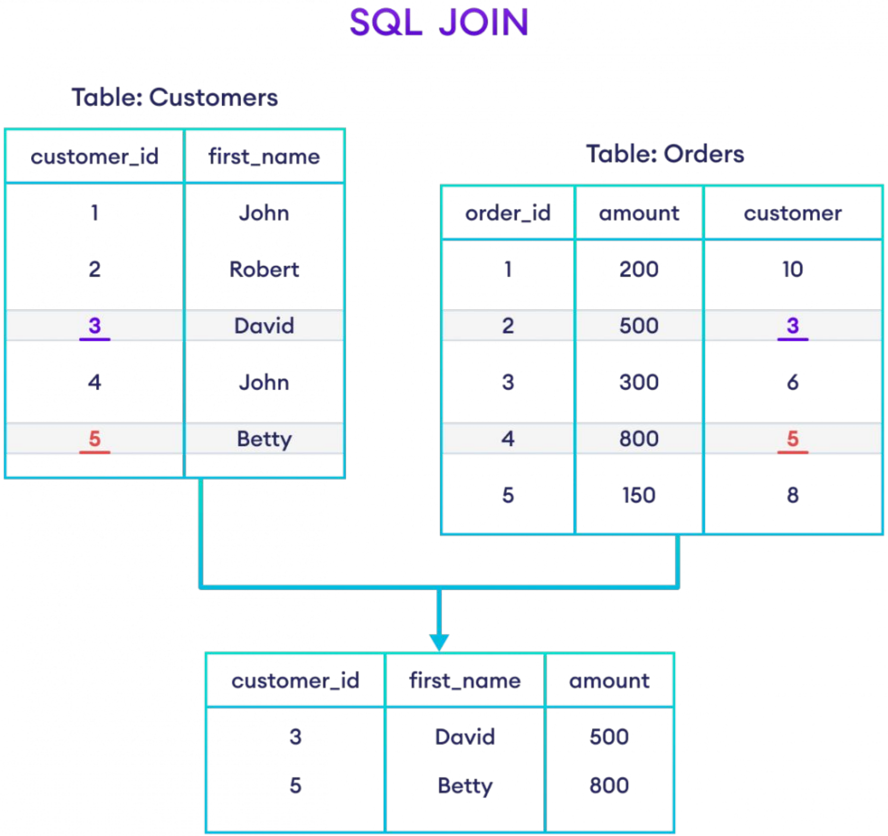
SQL JOINS

We join by using a **Primary Key** from one table that is stored as a **Foreign Key** in another table

Here - the Customers Table has a customer_id, this is the **primary key** because it uniquely identifies each User

The Orders table has the **foreign key customer** because it relates to a **foreign table: customer**

We connect the primary key ID on Customers to Orders to get Customer Info on the order



SQL JOIN

Table: Customers

customer_id	first_name
1	John
2	Robert
3	David
4	John
5	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	3
3	300	6
4	800	5
5	150	8

customer_id	first_name	amount
3	David	500
5	Betty	800

Subqueries

Putting queries inside of your queries

Queries can get complicated, especially as we try to do analysis on our SQL data

Due to the way SQL works, we might not be able to do queries all at once;

For instance, we can’t refer to some calculations that are made in the query later on in the query

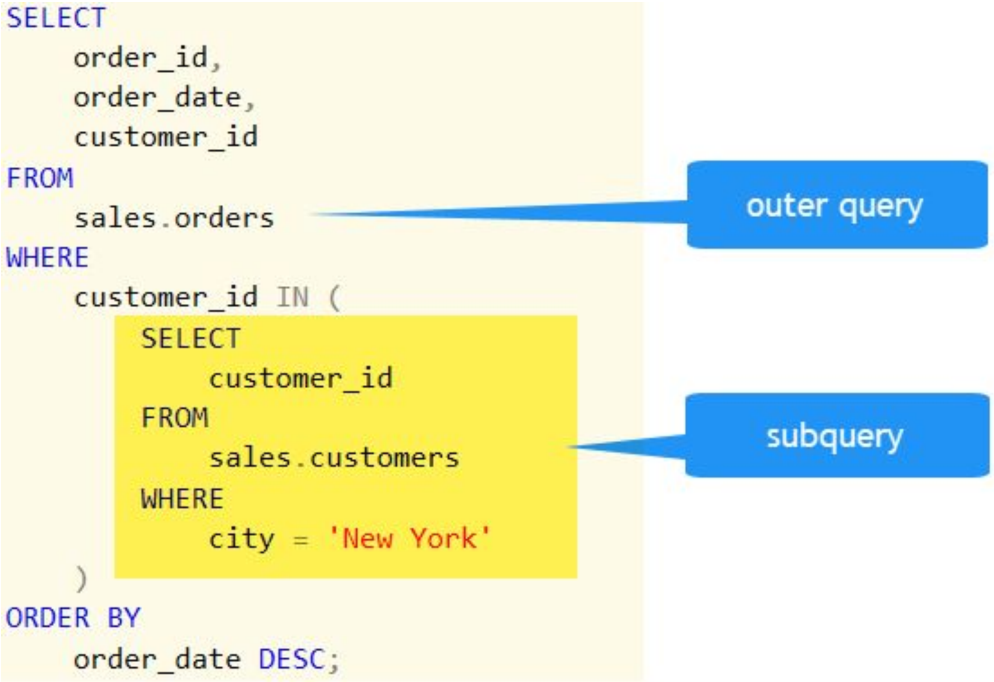
To get around that, we can create “*subqueries*”

Subquery Construction

Subqueries are smaller queries that **we put within another query**

By breaking up a larger query into smaller components, we can do more advanced queries;

realistically, we can’t go over all the uses of a subquery but we can go over where you can put them

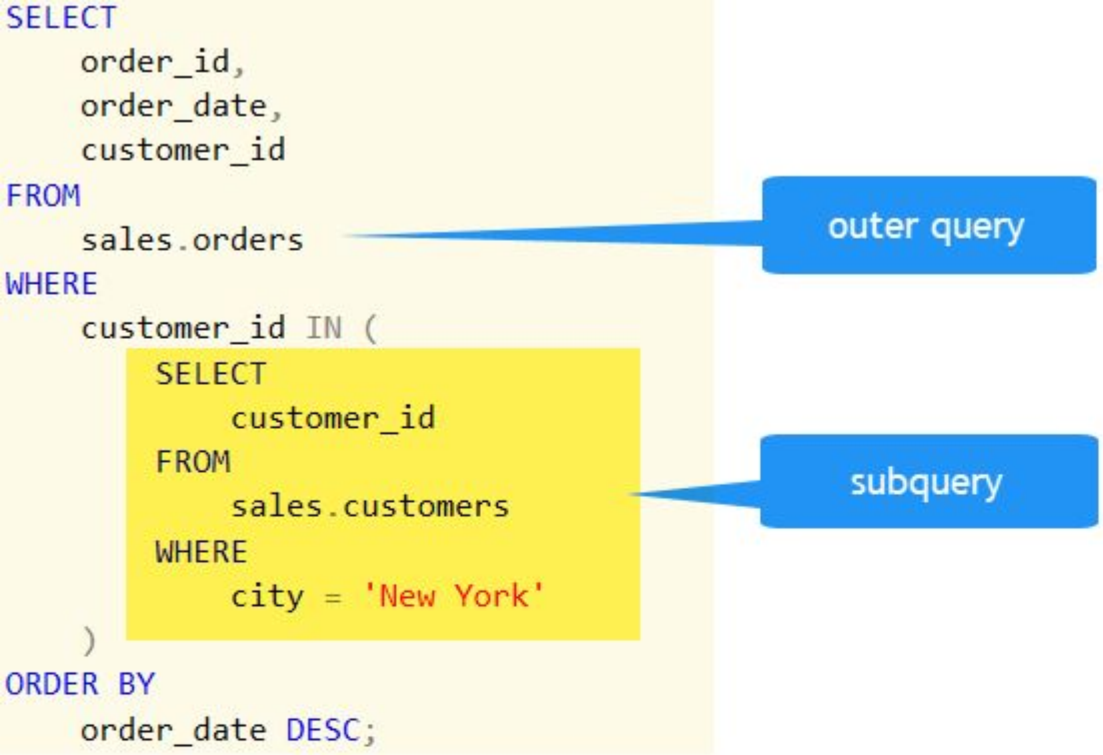


```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

Subquery in WHERE

One of the uses for a subquery, could be filtering a larger table by putting it in a WHERE clause

We use the subquery to generate a list of customers in New York we’re interested in then use that result to query orders for customers in that result



```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

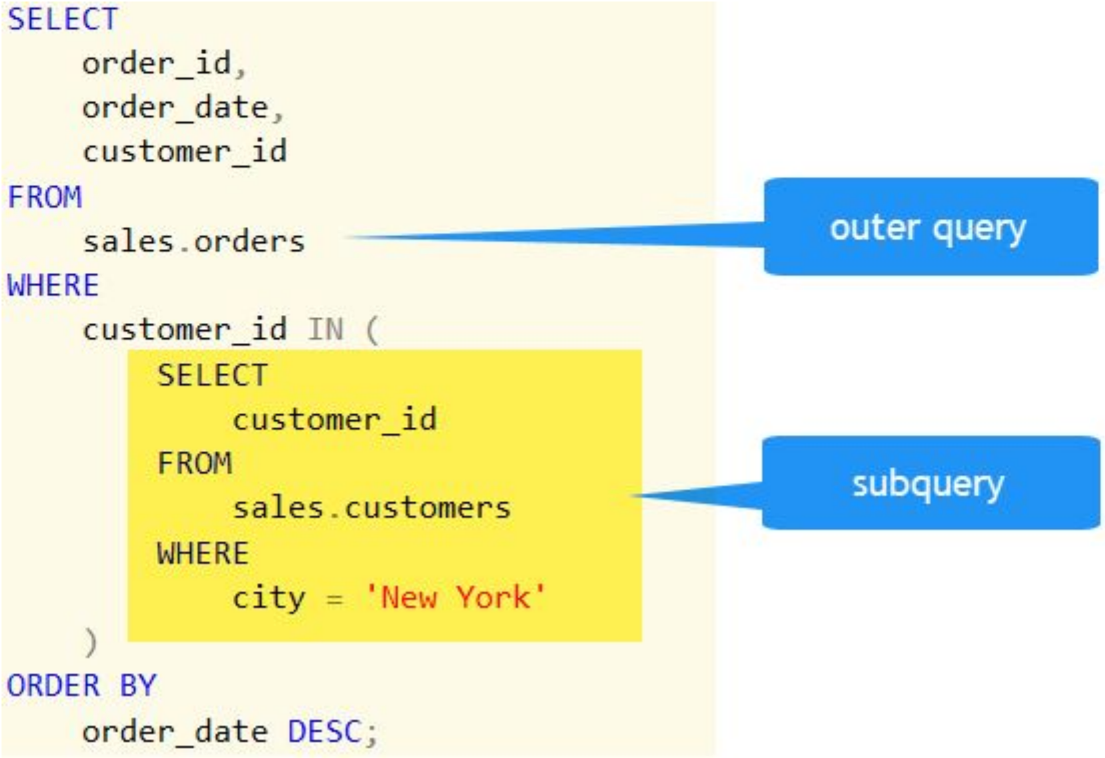
Subquery in WHERE

WHERE clauses:

SELECT *

FROM outer

WHERE outer_feature IN (subquery)



```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

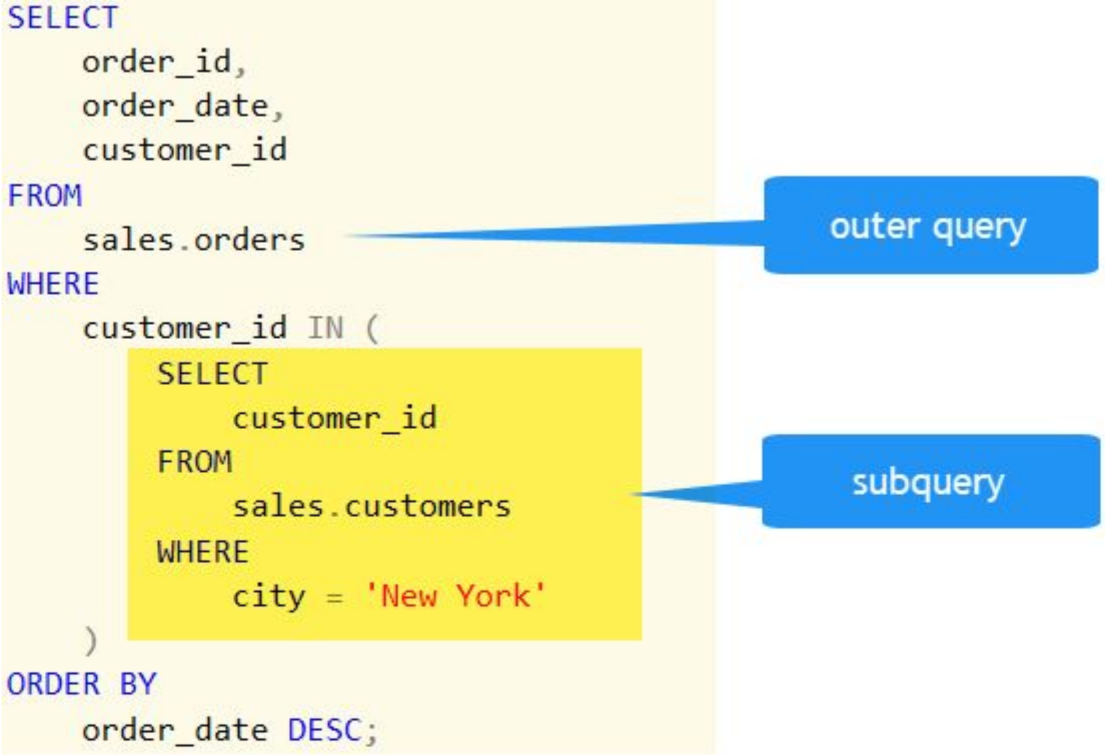
Subquery in WHERE

WHERE clauses example:

SELECT id, username

FROM users

WHERE id IN (SELECT id FROM likes WHERE post_id = 1215)



```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

Subquery in FROM

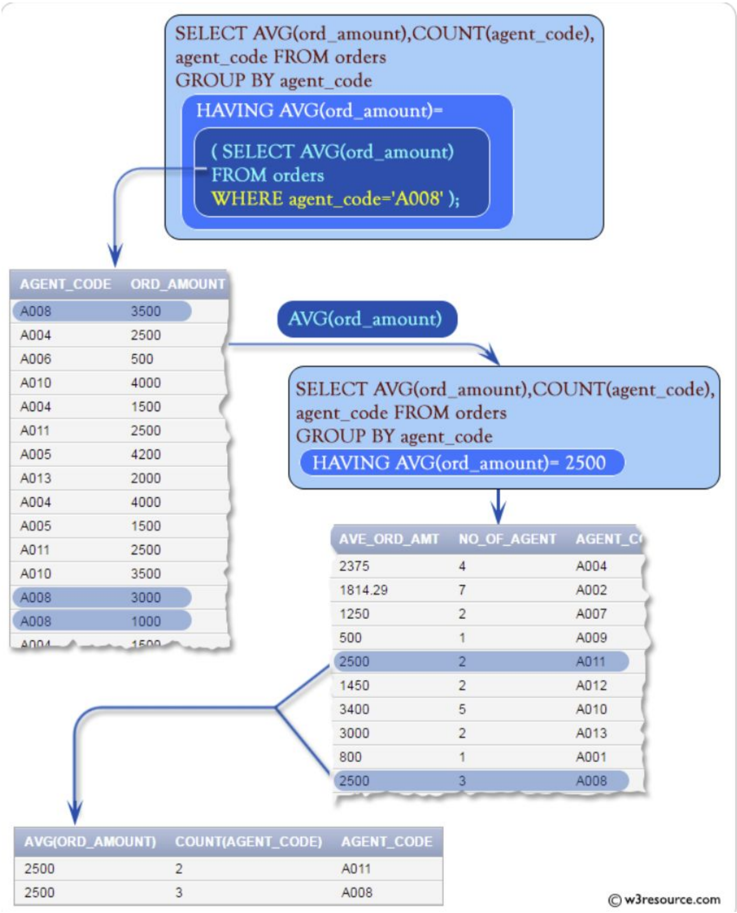
```
-- Find the posts with more likes than the average number of likes

SELECT AVG(total_likes)

FROM (

SELECT post_id, COUNT(id) as total_likes

FROM likes

WHERE post_id IS NOT NULL

GROUP BY post_id

ORDER BY total_likes DESC

) as like_counts;
```

Subquery in HAVING

Much like the WHERE clause, we can put a subquery inside of a HAVING clause to filter our grouped data

Note: this usually means that the result must be a singular item and not a list of items

```
SELECT
AVG(ord_amount),COUNT(agent_code),
agent_code FROM orders
GROUP BY agent_code
HAVING AVG(ord_amount)=2500;
```



```
SELECT AVG(ord_amount),COUNT(agent_code),
agent_code FROM orders
GROUP BY agent_code
HAVING AVG(ord_amount)=
  ( SELECT AVG(ord_amount)
  FROM orders
  WHERE agent_code='A008' );
```

AGENT_CODE	ORD_AMOUNT
A008	3500
A004	2500
A006	500
A010	4000
A004	1500
A011	2500
A005	4200
A013	2000
A004	4000
A005	1500
A011	2500
A010	3500
A008	3000
A008	1000
A004	1500

```
SELECT AVG(ord_amount),COUNT(agent_code),
agent_code FROM orders
GROUP BY agent_code
HAVING AVG(ord_amount)= 2500
```

AVE_ORD_AMT	NO_OF_AGENT	AGENT_C
2375	4	A004
1814.29	7	A002
1250	2	A007
500	1	A009
2500	2	A011
1450	2	A012
3400	5	A010
3000	2	A013
800	1	A001
2500	3	A008

AVG(ORD_AMOUNT)	COUNT(AGENT_CODE)	AGENT_CODE
2500	2	A011
2500	3	A008

© w3resource.com

Subquery in FROM

Sometimes we want to do some pre-processing on a query or more complicated math that we can’t do in a normal query; we can query a subquery directly

This isn’t *super* recommended as there are probably better ways to do it but it is good for simple cases

```
SELECT MAX(new_col)

FROM (

SELECT  CONCAT(col1, col2, col3) as new_col

FROM table

)
```

CASE Statements

CASE statements

SQL CASE statements should be thought of as “if” statements, they are even written with similar language

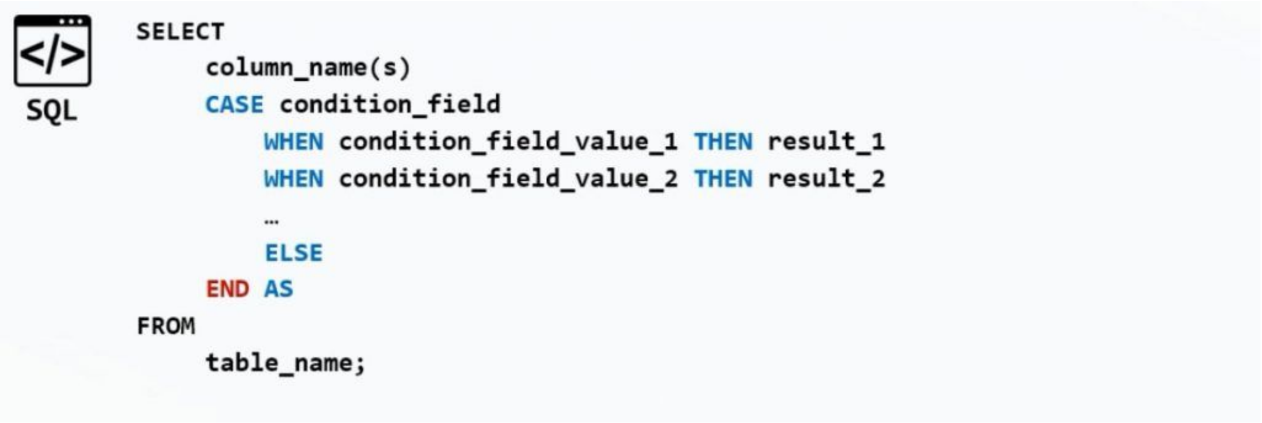
CASE (optional field)

WHEN THEN

WHEN THEN

ELSE final_option

END



```
SELECT
    column_name(s)
    CASE condition_field
        WHEN condition_field_value_1 THEN result_1
        WHEN condition_field_value_2 THEN result_2
        …
        ELSE
    END AS
FROM
    table_name;
```

CASE statements

CASE statements can be a great way for restructuring values for instance, maybe we want to bucket our customers based on their spending

```
SELECT customer,
CASE
WHEN spending >= 1000 THEN ‘high spender’
WHEN spending  >= 100 THEN ‘low spender’
ELSE ‘no spender’
END as spender_bucket
```

CASE statements

There is a special usage of a case statement if we’re interested in simple cases, we can select the column and just note the values we are curious about

```
SELECT item,
CASE color
WHEN 1 THEN ‘red’
WHEN 2 THEN ‘blue’
WHEN 3 THEN ‘green’
ELSE ‘No color’
END as color
```

```
SELECT item,
CASE
WHEN color = 1 THEN ‘red’
WHEN color = 2 THEN ‘blue’
WHEN color = 3 THEN ‘green’
ELSE ‘No color’
END as color
```

CASE statements

```
SELECT id, username,
  CASE
    WHEN post_count = 0 THEN 'Inactive User'
    WHEN post_count = 1 THEN 'Casual User'
    WHEN post_count <= 5 THEN 'Regular User'
    ELSE 'Active User'
  END AS user_category
FROM (
  SELECT
    users.id,
    users.username,
    COUNT(posts.id) AS post_count
  FROM users
  LEFT JOIN posts ON users.id = posts.user_id
  GROUP BY users.id, users.username
) AS UserPostCounts;
```

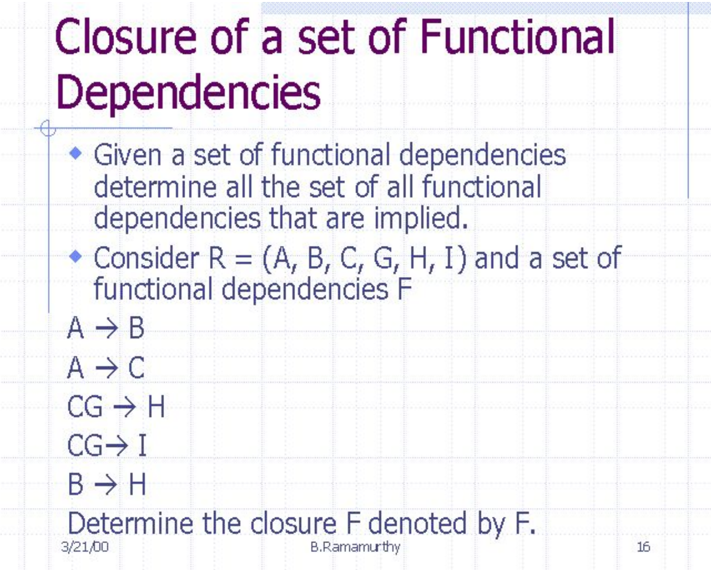
DB Normalization

Normalization

While the concept of normalization is rooted in mathematical notation named **functional dependency**, we will **not** delve into this.

Instead, we will delve into practical considerations.

NOTE: There’s no harm in exploring the maths of this!



Closure of a set of Functional Dependencies

- Given a set of functional dependencies determine all the set of all functional dependencies that are implied.
- Consider R = (A, B, C, G, H, I) and a set of functional dependencies F

A → B

A → C

CG → H

CG→ I

B → H

Determine the closure F denoted by F.

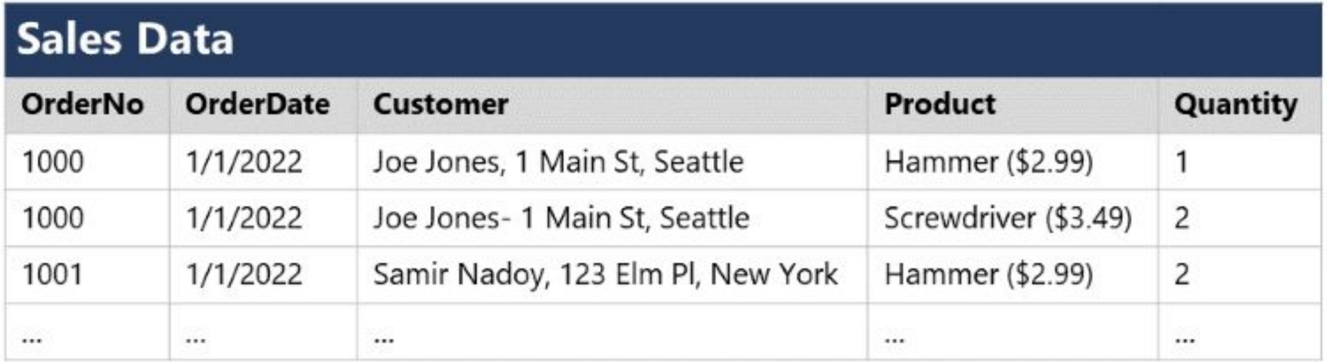
3/21/00 B.Ramamurthy 16

Normalization

Having a **normalized database is excellent for data storage** (not for analytics however). Normalizing a database involves the following steps:

1. *Separate each entity into its own table*
2. *Separate each discrete attribute into its own column.*
3. *Uniquely identify each entity instance (row) using a primary key.*
4. *Use foreign key columns to link related entities.*

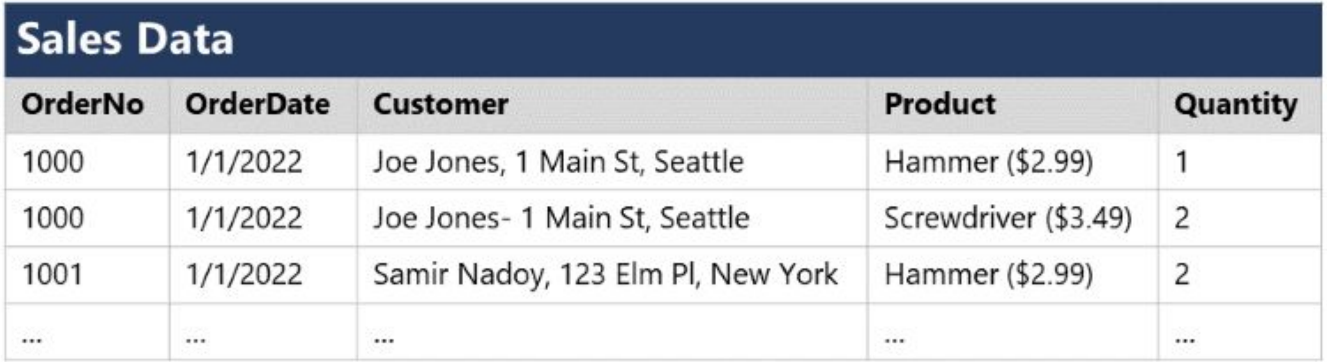
Can you see why these rules are good for data storage?



Sales Data

OrderNo	OrderDate	Customer	Product	Quantity
1000	1/1/2022	Joe Jones, 1 Main St, Seattle	Hammer ($2.99)	1
1000	1/1/2022	Joe Jones- 1 Main St, Seattle	Screwdriver ($3.49)	2
1001	1/1/2022	Samir Nadoy, 123 Elm Pl, New York	Hammer ($2.99)	2
...	...	...	...	...

For example, take a look at the following table. If I asked you to update the **price of a hammer**, what would you have to do?

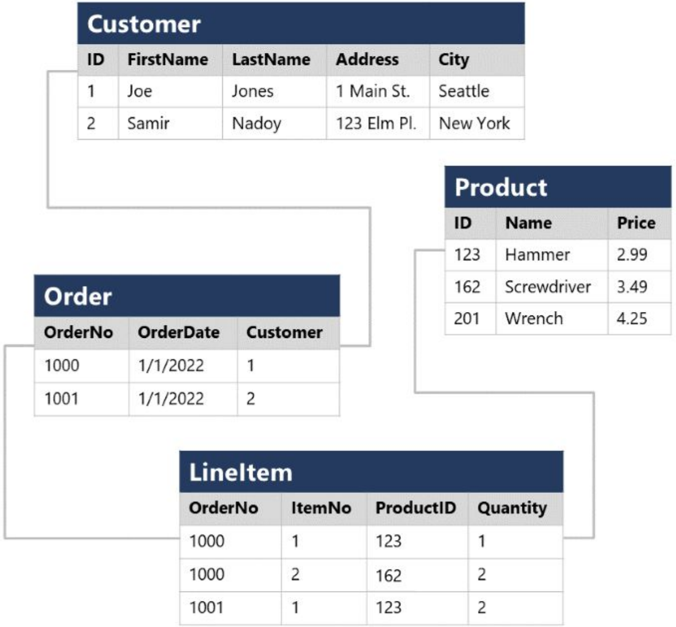


Sales Data

OrderNo	OrderDate	Customer	Product	Quantity
1000	1/1/2022	Joe Jones, 1 Main St, Seattle	Hammer ($2.99)	1
1000	1/1/2022	Joe Jones- 1 Main St, Seattle	Screwdriver ($3.49)	2
1001	1/1/2022	Samir Nadoy, 123 Elm Pl, New York	Hammer ($2.99)	2
...	...	...	...	...

You would have to find every single **Product** that's a hammer and do some non-trivial coding to extract the price and change it. This will take **multiple operations!**

Remember: laziness is prioritized in engineering.



Customer

ID	FirstName	LastName	Address	City
1	Joe	Jones	1 Main St.	Seattle
2	Samir	Nadoy	123 Elm Pl.	New York

Product

ID	Name	Price
123	Hammer	2.99
162	Screwdriver	3.49
201	Wrench	4.25

Order

OrderNo	OrderDate	Customer
1000	1/1/2022	1
1001	1/1/2022	2

LineItem

OrderNo	ItemNo	ProductID	Quantity
1000	1	123	1
1000	2	162	2
1001	1	123	2

Attributes are split into their own column

Entities exist in their own table

Rows contain unique primary keys

Foreign keys link related entities

Instead, we normalize this database by redesigning it in such a way that updating a hammer is simple!

Only one operation now! (***UPDATE product SET Price=3.99 WHERE Name=Hammer***)

Like always, there are both pros and cons

Normalization - Pros

- Reduces data **redundancy**
- Reduces **storage** requirements
- **Better security** by isolating tables
- Improves data integrity! Less rows to update, **less chance data will be incorrectly updated/lost.**

Normalization - Cons

- Reduces **speed**, data needs be **joined**
- Increases **complexity**, more maintenance
- Does **not satisfy** analytical/aggregation structures as data structures are inflexible and slow to access

Review Questions

1) Is this table normalized?

movie	actors	genre	budget(USD)
Oppenheimer	Cillian Murphy, Florence Pugh	Drama	100,000,000
Barbie	Margot Robbie, Ryan Gosling	Comedy	145,000,000

Review Questions

2) How would you normalize this table?

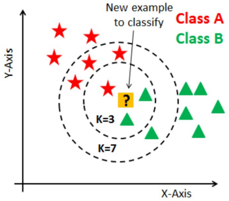
movie	actors	genre	budget(USD)
Oppenheimer	Cillian Murphy, Florence Pugh	Drama	100,000,000
Barbie	Margot Robbie, Ryan Gosling	Comedy	145,000,000

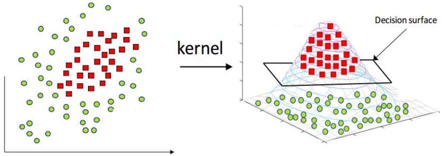
SQL Lab - SQL Bolt & Murder Mystery

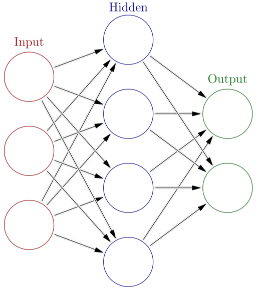
SQL Bolt

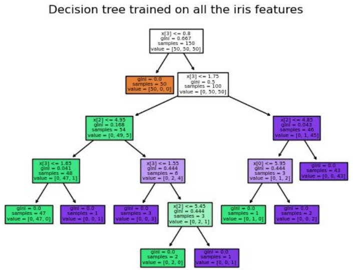
For the remaining lab time, work through SQL Bolt (Lessons 9 - 12).

Once you're done with that, check out the SQL Murder Mystery.









Wednesday

More SQL Practice!

- *SQL Leetcode Questions*