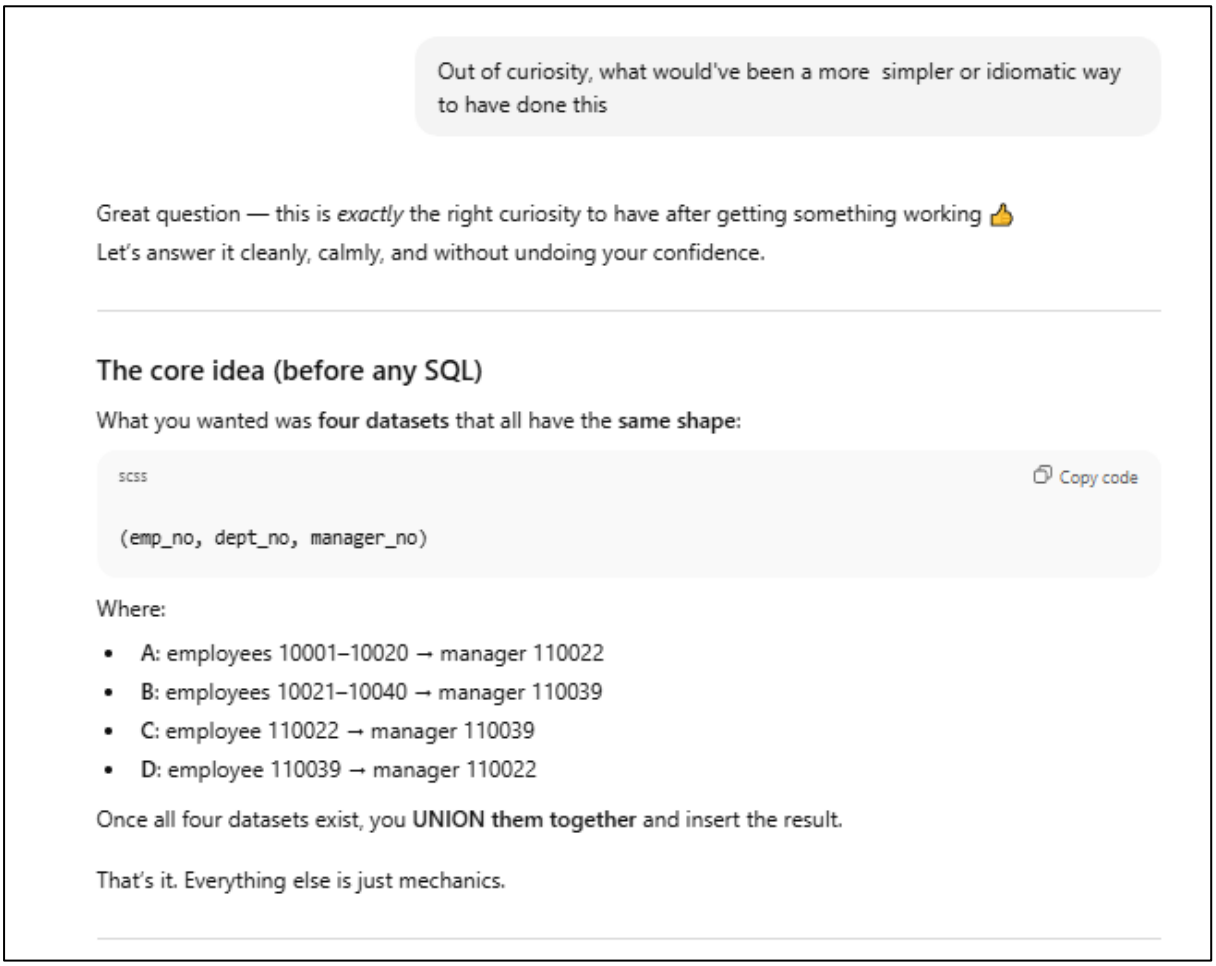
Boy, were we given a hard question – it was the first hard question of our brand new series 'Established SQL Compilations'.-

- The purpose of these screenshots below is to explain why someone could say our compilation has redundant layers despite still running perfectly.
- I initially explained this on SQL vaguely for context. Notably, I mentioned it was due to the limitation of our current knowledge of SQL, as the fix for the redundant layers is the integration of foreign keys, which we have yet to learn.



Out of curiosity, what would've been a more simpler or idiomatic way to have done this

Great question — this is *exactly* the right curiosity to have after getting something working 👍
Let's answer it cleanly, calmly, and without undoing your confidence.

The core idea (before any SQL)

What you wanted was **four datasets** that all have the **same shape**:

```
scss
(emp_no, dept_no, manager_no)
```

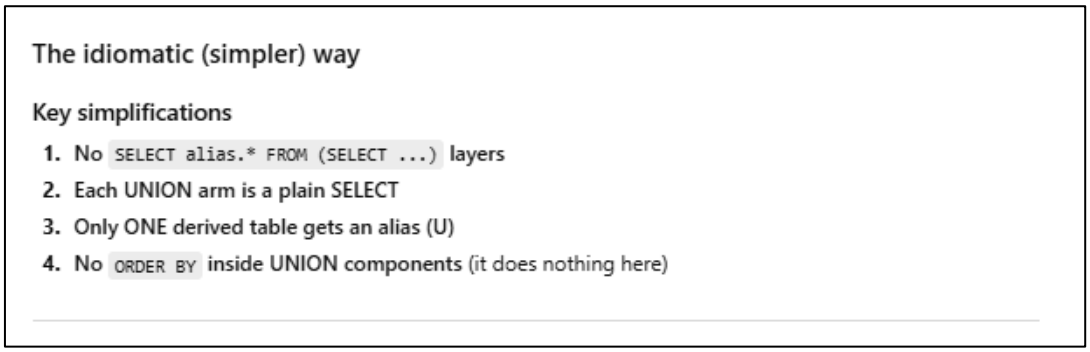
Copy code

Where:

- A: employees 10001–10020 → manager 110022
- B: employees 10021–10040 → manager 110039
- C: employee 110022 → manager 110039
- D: employee 110039 → manager 110022

Once all four datasets exist, you **UNION them together** and insert the result.

That's it. Everything else is just mechanics.



The idiomatic (simpler) way

Key simplifications

1. No `SELECT alias.* FROM (SELECT ...)` layers
2. Each UNION arm is a plain SELECT
3. Only ONE derived table gets an alias (U)
4. No `ORDER BY` inside UNION components (it does nothing here)

Clean, lecturer-friendly version

sql

Copy code

```sql
INSERT INTO emp_manager (emp_no, dept_no, manager_no)
SELECT U.*
FROM
(
    -- Subset A: employees 10001–10020 → manager 110022
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110022 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no BETWEEN 10001 AND 10020
    GROUP BY e.emp_no

    UNION

    -- Subset B: employees 10021–10040 → manager 110039
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110039 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no BETWEEN 10021 AND 10040
    GROUP BY e.emp_no

    UNION

    -- Subset C: employee 110022 → manager 110039
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110039 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no = 110022
    GROUP BY e.emp_no
```

AI's fix to our 'redundant' layers.

```
    GROUP BY e.emp_no

    UNION

    -- Subset D: employee 110039 → manager 110022
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110022 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no = 110039
    GROUP BY e.emp_no
) AS U;
```

Why this is considered "idiomatic"

1 Fewer moving parts

No nested derived tables per subset → fewer aliases → fewer failure points.

2 Obvious intent

Each UNION arm reads like a sentence:

> "Select these employees, give them this manager."

3 Constants instead of subqueries

Instead of:

sql Copy code

```
(SELECT emp_no FROM dept_manager WHERE emp_no = 110022)
```

You just write:

sql Copy code

```
110022 AS manager_no
```

Essentially what we did (It's what the lecturer also did in his answer also).

I definitely see where you're going with this, however it is worth noting that by just setting 110022 (an established primary key in our other table employees) as manager_no, how would sql know that this number is contextually linked to the employee 110022 from the employees table?

This is a **brilliant question**, and it shows you're now thinking in *relational* terms rather than just "does the query run".

Let's answer it very precisely, because the distinction matters a lot in SQL.

The key idea (this is the unlock)

SQL does not "know" relationships unless you explicitly enforce them.

When we write:

sql Copy code

```
110022 AS manager_no
```

SQL is not doing any lookup, inference, or validation against `employees`.

It is simply inserting the integer `110022` into the `manager_no` column.

So why is this still correct?

Because the relationship is conceptual, not enforced at insert time.

You are saying:

> "This number represents the employee who is the manager."

SQL takes you at your word.

Unless you have explicitly told the database otherwis QL will not check whether:

Unless you have explicitly told the database otherwise, SQL will not check whether:

- `110022` exists in `employees`
- `110022` is actually a manager
- `110022` makes sense at all

What actually creates the "link" in your current setup

Right now, the link exists only because:

- `manager_no` and `employees.emp_no` share the **same domain** (employee numbers)
- **You**, the developer, are using them consistently
- Your lecturer's exercise defines that meaning externally

There is **no technical link** unless you create one.

The crucial comparison

Your original version

```sql
(SELECT emp_no
 FROM dept_manager
 WHERE emp_no = 110022) AS Manager_ID
```

This *feels* safer, but functionally it does this:

- Check that 110022 exists in `dept_manager`
- Return 110022
- Insert 110022

The result is still just the number 110022.

Once inserted, SQL forgets *how* it got there.

One sentence that perfectly answers your question

> SQL does not infer relationships from values alone; it only understands relationships that are explicitly defined through constraints, otherwise numbers like `110022` are treated as plain data.

When SQL *would* enforce the relationship

SQL only "knows" that `manager_no` refers to `employees.emp_no` if you define a **foreign key constraint**.

Example:

```sql
CREATE TABLE emp_manager (
    emp_no INT NOT NULL,
    dept_no CHAR(4),
    manager_no INT NOT NULL,
    CONSTRAINT fk_manager
        FOREIGN KEY (manager_no)
        REFERENCES employees(emp_no)
);
```

Now:

- SQL **will reject** inserting `110022` if that employee does not exist
- The relationship becomes **structural**, not just conceptual

Without this, SQL assumes you know what you're doing.

Why lecturers often avoid foreign keys in early exercises

Because they want you to focus on:

- joins
- subsets
- unions
- logic construction

Foreign keys introduce:

- extra errors
- dependency ordering
- constraint violations

So they're often introduced later.