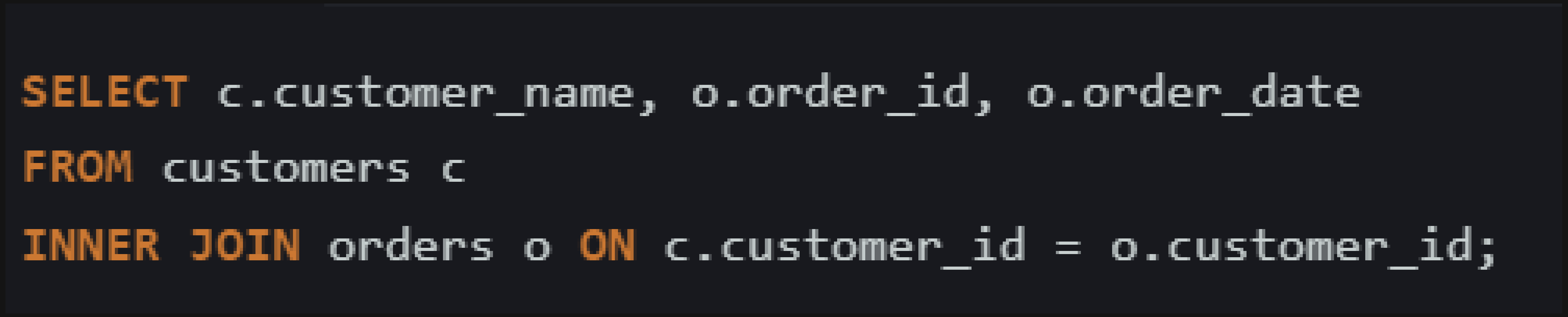
SQL Series

HOW TO USE
JOIN CLAUSE

Types of Joins

INNER JOIN

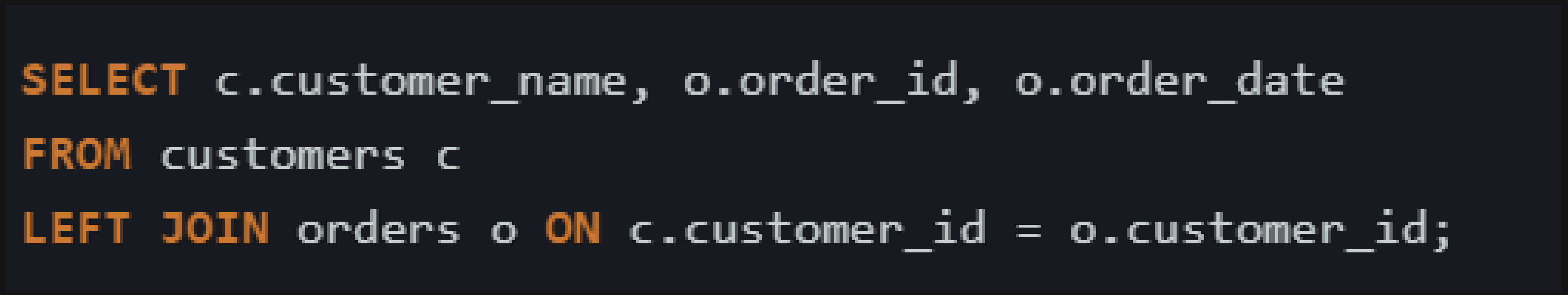
📌 *Returns rows where the join condition matches in both tables.*



```sql
SELECT c.customer_name, o.order_id, o.order_date
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

LEFT JOIN

📌 *Returns all rows from the left table, even if there's no match in the right table.*

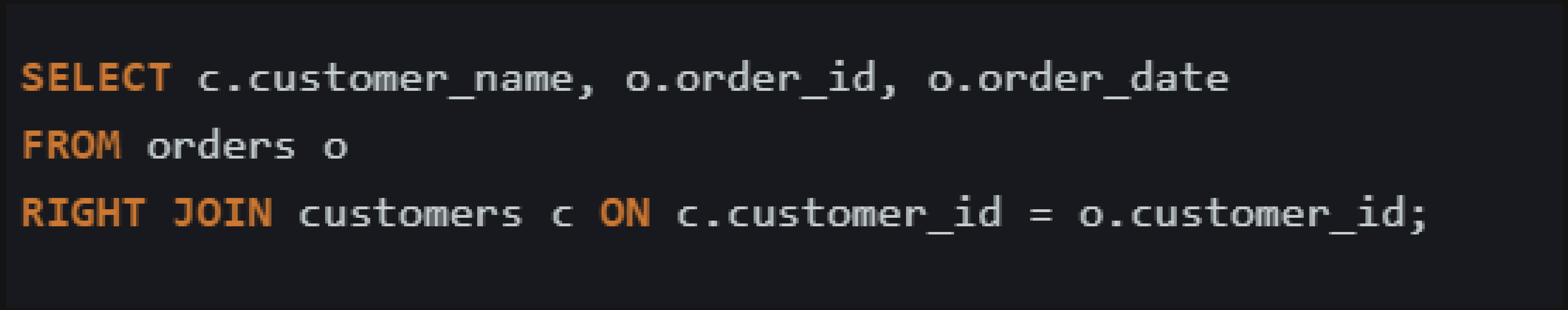


```sql
SELECT c.customer_name, o.order_id, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

Types of Joins

RIGHT JOIN

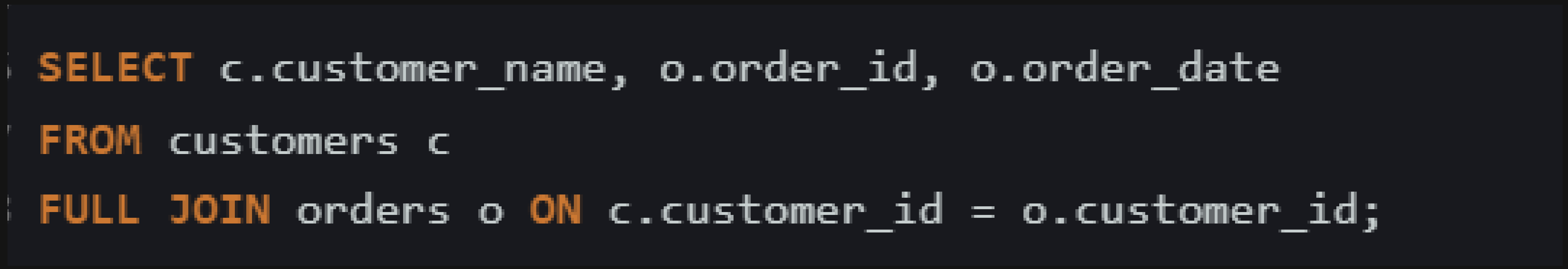
📌 *Returns all rows from the right table, even if there's no match in the left table..*



```sql
SELECT c.customer_name, o.order_id, o.order_date
FROM orders o
RIGHT JOIN customers c ON c.customer_id = o.customer_id;
```

FULL JOIN

📌 *Returns all rows from both tables, regardless of whether there's a match.*

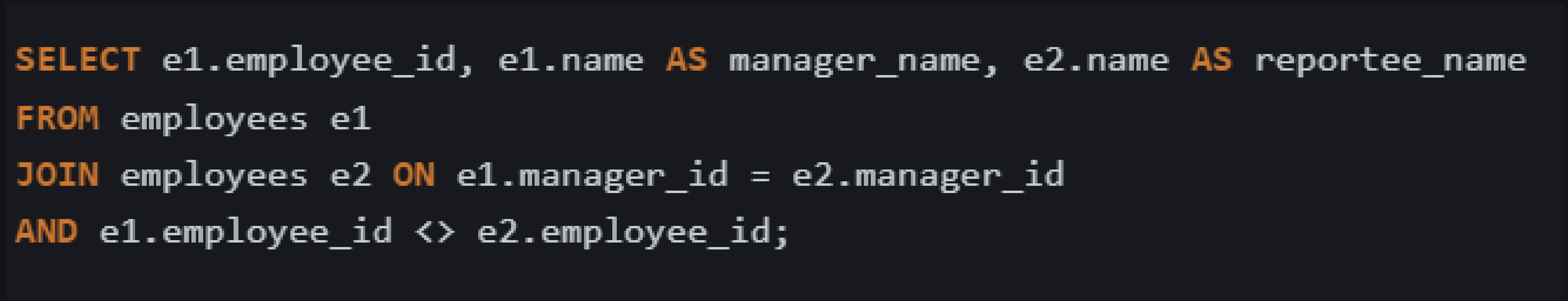


```sql
SELECT c.customer_name, o.order_id, o.order_date
FROM customers c
FULL JOIN orders o ON c.customer_id = o.customer_id;
```

Types of Joins

SELF JOIN

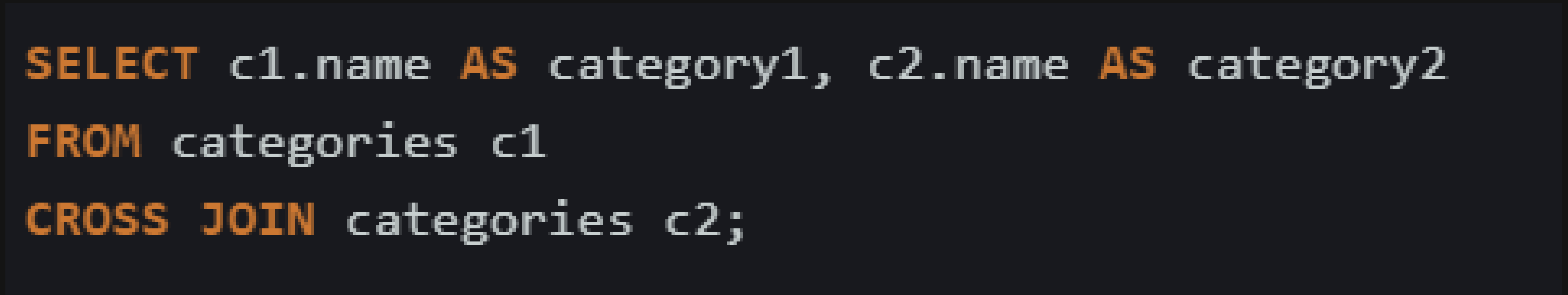
📌 *Allows us to join a table to itself, essentially treating it as two separate but identical tables*



```sql
SELECT e1.employee_id, e1.name AS manager_name, e2.name AS reportee_name
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.manager_id
AND e1.employee_id <> e2.employee_id;
```

CROSS JOIN

📌 *Combines every row from one table with every row from another table, without any specific join condition.*



```sql
SELECT c1.name AS category1, c2.name AS category2
FROM categories c1
CROSS JOIN categories c2;
```

Advantages:

📌**Retrieve data from multiple tables**: Access related information across tables in a single query.

📌 **Simplify complex queries:** Reduce the need for multiple separate queries.

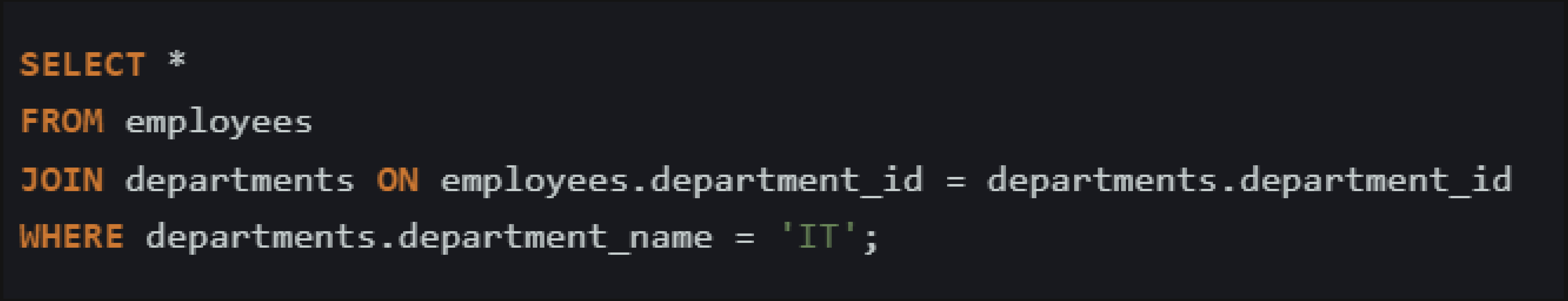
📌**Create richer datasets:** Combine data from different tables to unlock new insights.

📌**Improve data analysis:** Gain a more comprehensive understanding of your data.

Basic Syntax:

📌*JOIN clause is used to combine rows from two or more tables based on a related column between them.*

📌*Example: Suppose we have two tables, 'employees' and 'departments.' To retrieve information about employees in the 'IT' department*

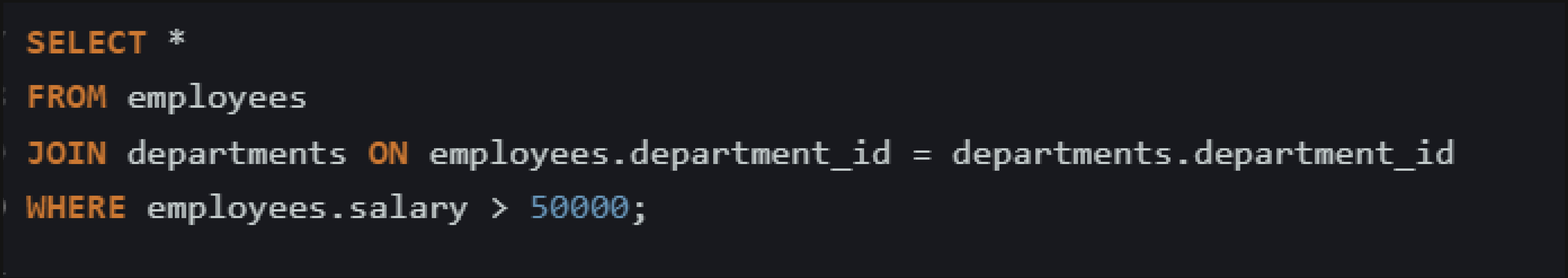


```
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE departments.department_name = 'IT';
```

Comparison Operator:

📌 *We can use comparison operators in the ON clause to specify the condition for joining the tables.*

📌*Example: to find employees with a salary greater than $50,000, you might use*

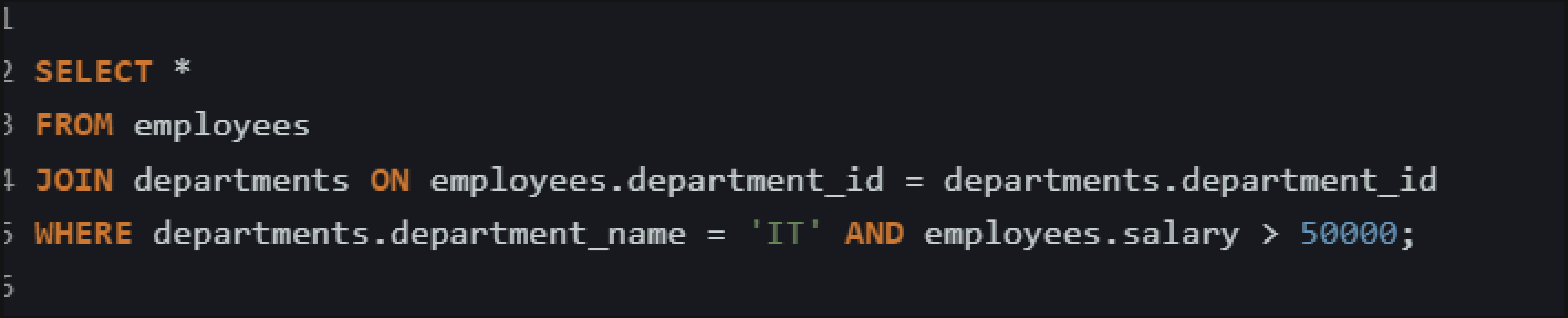


```
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE employees.salary > 50000;
```

Logical Operator:

📌 *Logical operators such as AND or OR can be used to create more complex conditions for joining.*

📌*Example: To find employees in the 'IT' department with a salary greater than $50,000, you can use*

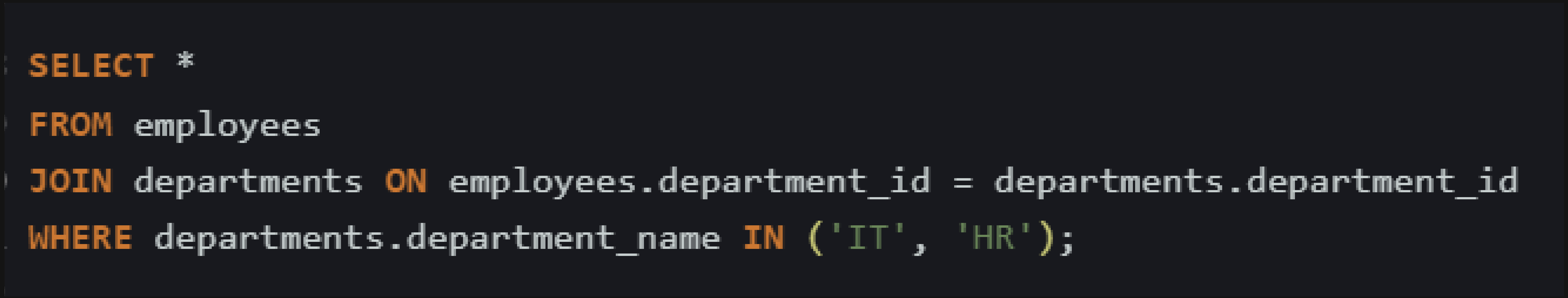


```sql
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE departments.department_name = 'IT' AND employees.salary > 50000;
```

IN Clause:

📌 *The IN clause is useful when you want to match a column against multiple values*

📌*Example: Retrieve employees from departments 'IT' or 'HR':*

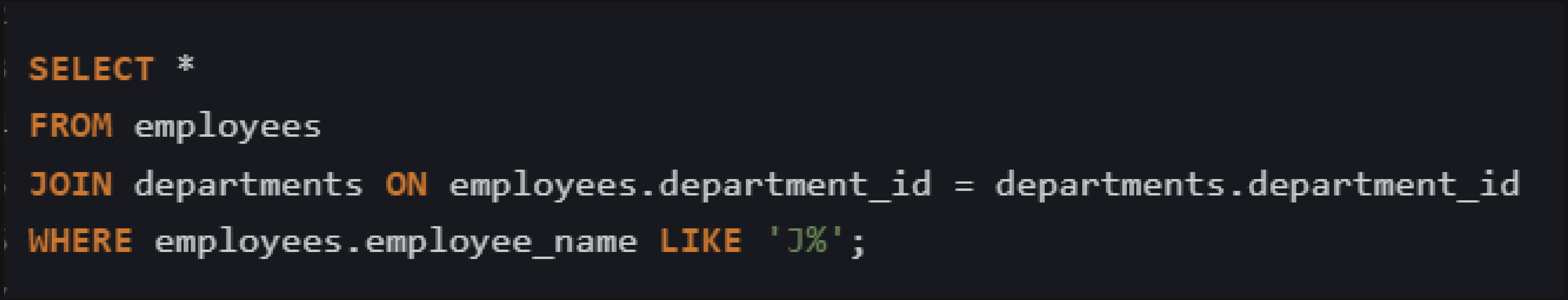


```
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE departments.department_name IN ('IT', 'HR');
```

Wildcard:

📌**Wildcards like % can be used to match patterns in string data.**

📌*Example: Find employees whose names start with 'J':*

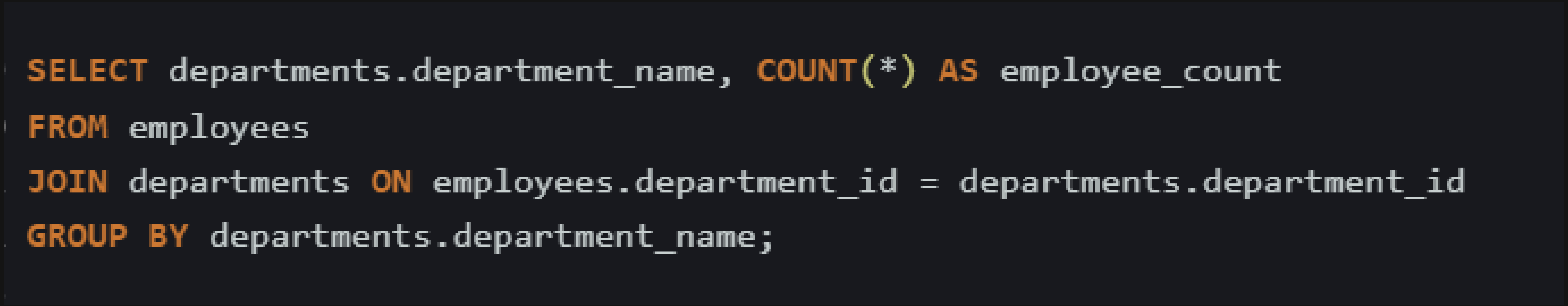


```sql
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE employees.employee_name LIKE 'J%';
```

GROUP BY:

📌 *The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows.*

📌*Example: Get the count of employees in each department:*

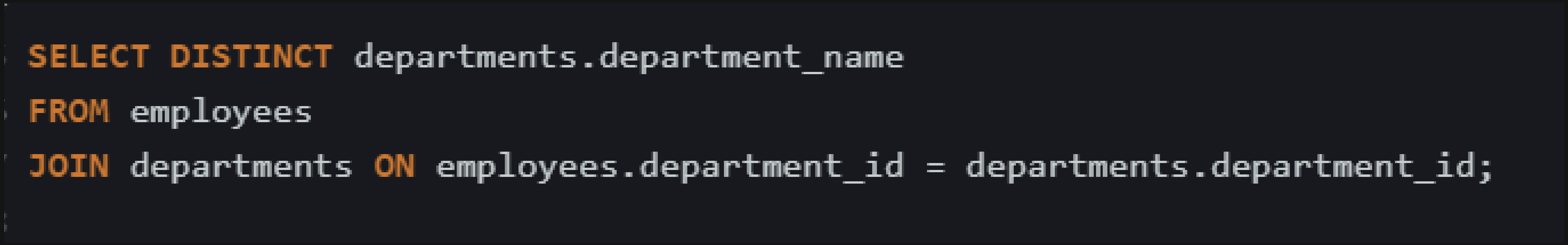


```sql
SELECT departments.department_name, COUNT(*) AS employee_count
FROM employees
JOIN departments ON employees.department_id = departments.department_id
GROUP BY departments.department_name;
```

DISTINCT:

📌 *The DISTINCT keyword is used to retrieve unique values from a column.*

📌*Example: Retrieve distinct department names:*

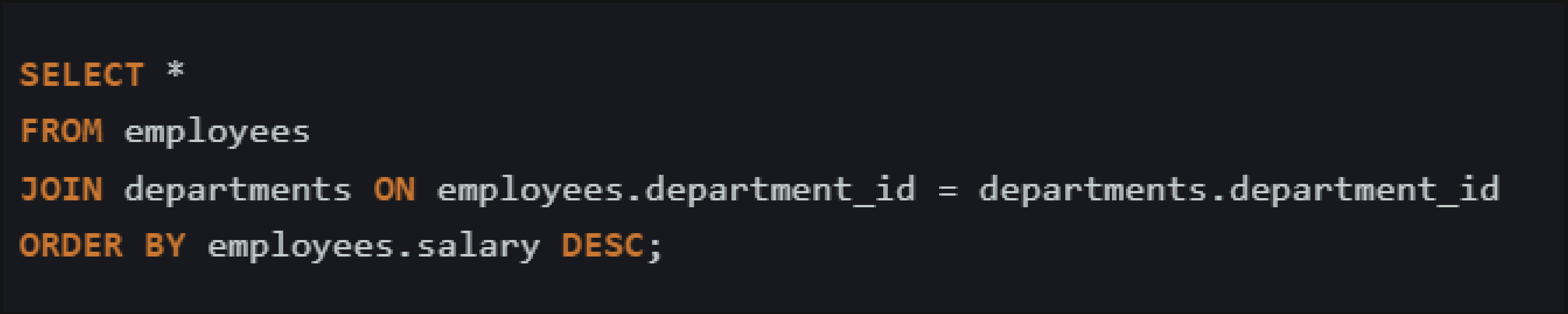


```
SELECT DISTINCT departments.department_name
FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

ORDER BY Clause:

📌 *The ORDER BY clause is used to sort the result set in ascending or descending order.*

📌*Example: Get employees sorted by salary in descending order:*

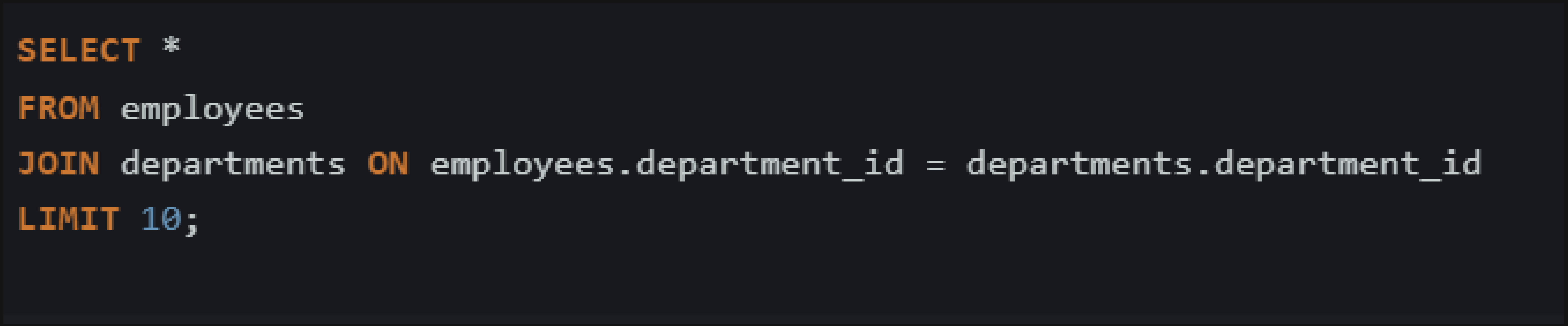


```sql
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
ORDER BY employees.salary DESC;
```

LIMIT Clause:

📌 *The LIMIT clause is used to limit the number of rows returned in a result set.*

📌*Example: Retrieve the first 10 employees:*

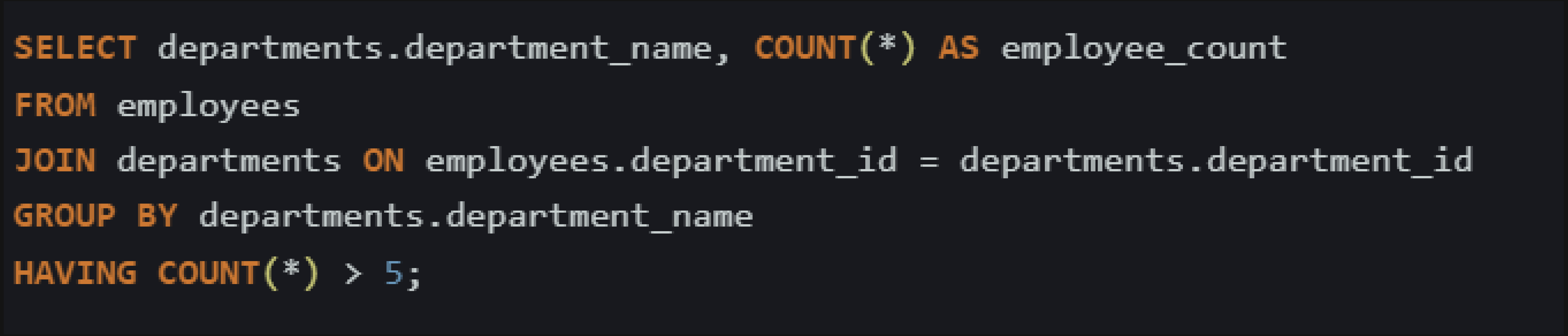


```
SELECT *
FROM employees
JOIN departments ON employees.department_id = departments.department_id
LIMIT 10;
```

HAVING Clause:

📌 *Window functions perform a calculation across a set of table rows related to the current row.*

📌*Example: Rank employees based on their salary within each department:*

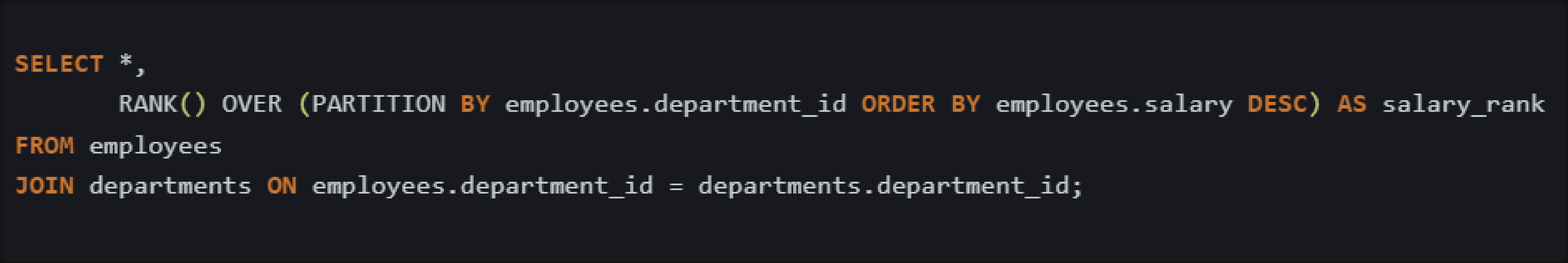


```sql
SELECT departments.department_name, COUNT(*) AS employee_count
FROM employees
JOIN departments ON employees.department_id = departments.department_id
GROUP BY departments.department_name
HAVING COUNT(*) > 5;
```

Window Function:

📌 *The HAVING clause is used with the GROUP BY clause to filter the results based on a specified condition.*

📌*Example: Get departments with more than 5 employees:*



```
SELECT *,
       RANK() OVER (PARTITION BY employees.department_id ORDER BY employees.salary DESC) AS salary_rank
FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

FOUND HELPFUL?

REPOST

https://www.linkedin.com/in/sabyasachi-bhattacharjee/