SQL: A Practical Guide with Code Examples

1. SQL Basics

SELECT, WHERE, ORDER BY, GROUP BY, HAVING

- **SELECT** retrieves specific columns:

```
SELECT name, age FROM employees;
```

- **WHERE** filters rows before grouping:

```
SELECT * FROM employees WHERE department = 'HR';
```

- **ORDER BY** sorts the output:

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

- **GROUP BY** groups rows with the same values for aggregate functions:

```
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

- **HAVING** filters groups, not individual rows:

```
SELECT department, COUNT(*) as count
FROM employees
GROUP BY department
HAVING count > 5;
```

Execution order: `WHERE` filters rows; `GROUP BY` forms groups; `HAVING` filters groups; `ORDER BY` sorts results[1][2].

2. Joins: INNER, LEFT, RIGHT, FULL OUTER

- **INNER JOIN**: Returns rows with matching values in both tables.

```
SELECT employees.name, departments.dept_name
FROM employees
INNER JOIN departments
  ON employees.dept_id = departments.id;
```

- **LEFT JOIN**: All left table rows and matched right table rows (NULL if no match).

```
SELECT e.name, d.dept_name
FROM employees e
LEFT JOIN departments d
  ON e.dept_id = d.id;
```

- **RIGHT JOIN**: All right table rows and matched left table rows (NULL if no match).

```
SELECT e.name, d.dept_name
FROM employees e
RIGHT JOIN departments d
  ON e.dept_id = d.id;
```

- **FULL OUTER JOIN**: All rows where there is a match in either table; NULLs where there is no match[3][4].

```
SELECT e.name, d.dept_name
FROM employees e
FULL OUTER JOIN departments d
  ON e.dept_id = d.id;
```

3. Subqueries & Nested Queries

- **Subquery**: Query within a query.

```
SELECT name FROM employees
WHERE department_id IN (
  SELECT id FROM departments WHERE location = 'NY'
);
```

- **Correlated Subquery**: Uses data from the outer query.

```
SELECT name
FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id =
e.department_id);
```

- **Types**: Non-correlated (runs independently), Correlated (depends on outer query)[5][6].

4. CTEs and Window Functions

- **CTE (Common Table Expression):**

```
WITH high_salary AS (
  SELECT name, salary FROM employees WHERE salary > 50000
)
SELECT * FROM high_salary;
```

- **ROW_NUMBER, RANK, DENSE_RANK**:

```
SELECT name, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
       RANK() OVER (ORDER BY salary DESC) AS rank,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM employees;
```

With **PARTITION BY** (e.g., ranking within departments):

```
SELECT name, department,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_num
FROM employees;
```

5. SQL for EDA: CASE WHEN & Aggregations

- **CASE WHEN** for conditional logic:

```
SELECT name,
       CASE
          WHEN salary >= 60000 THEN 'Highly Paid'
          WHEN salary >= 40000 THEN 'Average'
          ELSE 'Low'
```

```
       END AS salary_bracket
FROM employees;
```

- **Aggregations**: COUNT, SUM, AVG, MIN, MAX

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

- **Combining CASE with aggregations**:

```
SELECT department,
       COUNT(CASE WHEN salary > 50000 THEN 1 END) AS high_earners
FROM employees
GROUP BY department;
```

6. Indexing & Query Optimization

- **Creating an Index:**

```
CREATE INDEX idx_salary ON employees(salary);
```

- **Index Usage**: Indexes help speed up WHERE, ORDER BY, and JOINs where indexed columns are involved.
- **Best Practices**:
  - Index columns that appear in WHERE, JOIN, or ORDER BY clauses.
  - Avoid over-indexing (can slow down updates/inserts).
- **Example for Query Optimization:**

```
SELECT * FROM employees WHERE salary BETWEEN 60000 AND 90000;
```

This. uses an index on `salary` for fast range search[7][8].

7. Using SQL with Pandas (sqlite3, SQLAlchemy)

Read from SQL:

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('database.db')
df = pd.read_sql_query("SELECT * FROM employees", conn)
conn.close()
```

Write DataFrame to SQL:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///database.db')
df.to_sql('employees', engine, if_exists='replace')
```

Execute SQL via SQLAlchemy:

```
from sqlalchemy import text
with engine.connect() as conn:
    result = conn.execute(text("SELECT * FROM employees"))
    for row in result:
        print(row)
```

⁂

1.