SQL Cheat Sheet

This cheat sheet provides a quick reference for common SQL operations and functions, adapted to work with the Classic Models database structure.

The examples use tables such as products, orders, customers, employees, offices, orderdetails, productlines, and payments as shown in the database diagram.

This structure represents a model car business, so the examples have been tailored to fit this context.

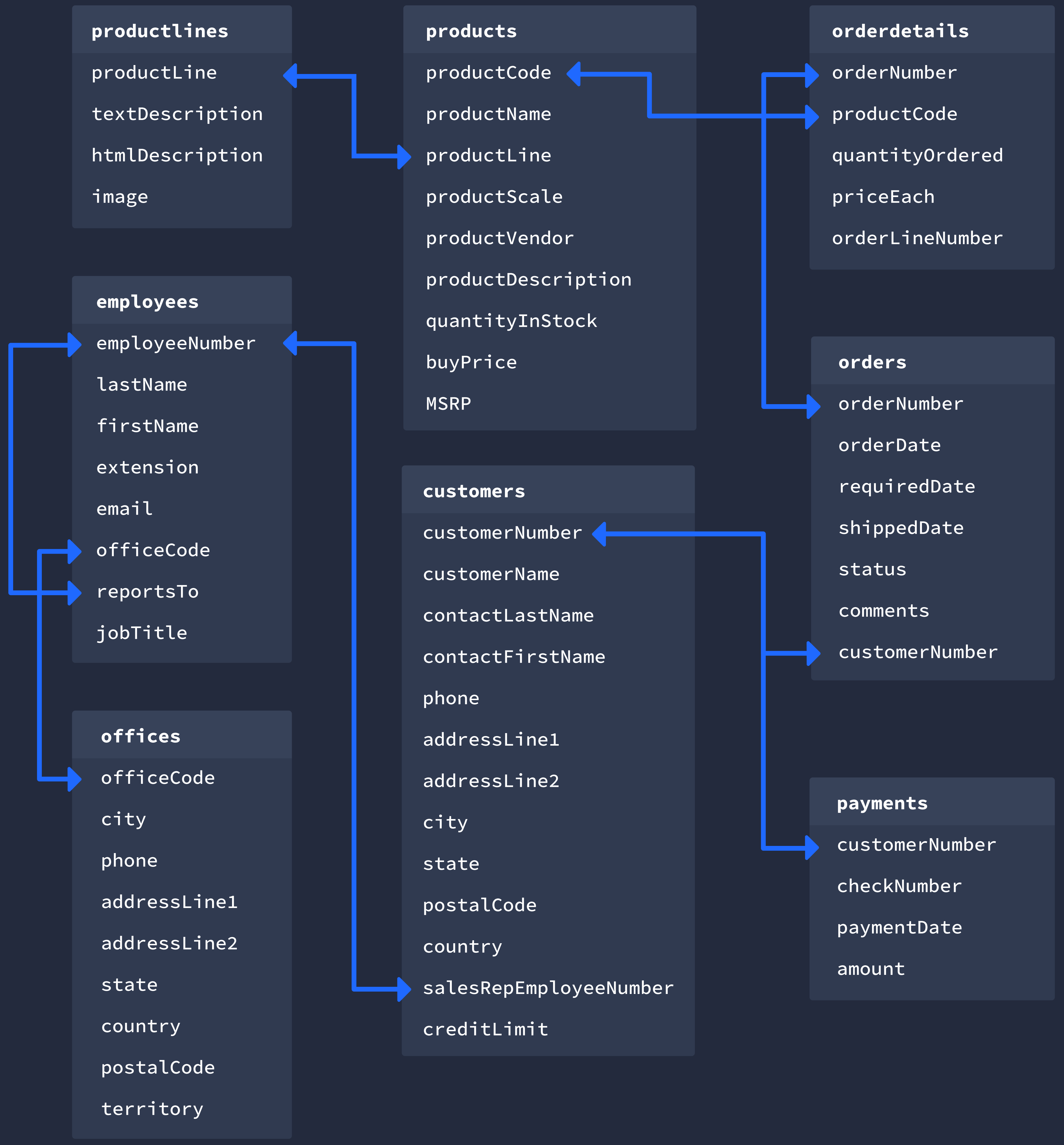

Table of Contents

Selection Queries

Clause	How to use	Explained
`SELECT`	`SELECT * FROM products;`	Display all columns from `products` table.
	`SELECT productName, buyPrice FROM products;`	Display only `productName` and `buyPrice` columns from `products` table.
`ORDER BY`	`SELECT productName, buyPrice FROM products ORDER BY buyPrice DESC;`	Sort the selected columns by `buyPrice` in descending order.
	`SELECT productName, buyPrice FROM products ORDER BY productName ASC;`	Sort the selected columns by `productName` in ascending order.
	`SELECT orderNumber, customerNumber, orderDate, status FROM orders ORDER BY customerNumber ASC, orderDate DESC;`	Sorts the data by `customerNumber` and then by `orderDate` within each `customerNumber`.
`DISTINCT`	`SELECT DISTINCT productLine FROM products;`	Retrieve unique values from `productLine` in `products` table.
	`SELECT DISTINCT city, country FROM customers ORDER BY country, city;`	Retrieve unique combinations of `city` and `country` where `customers` are located, sorted by `country` and then `city`.

Aggregate Functions

Clause	How to use	Explained
`SUM`	`SELECT SUM(quantityOrdered * priceEach) AS total_sales FROM orderdetails;`	Calculates the total sales from the `orderdetails` table.
`AVG`	`SELECT AVG(buyPrice) AS average_price FROM products;`	Averages the `buyPrice` values in `products`.
`ROUND`	`SELECT ROUND(AVG(buyPrice), 2) AS average_price FROM products;`	Rounds the average of `buyPrice` to two decimal places.
`MIN`	`SELECT MIN(buyPrice) AS lowest_price FROM products;`	Finds the minimum value in the `buyPrice` column of `products`.
`MAX`	`SELECT MAX(buyPrice) AS highest_price FROM products;`	Finds the maximum value in the `buyPrice` column of `products`.
`COUNT`	`SELECT COUNT(*) AS total_orders FROM orders;`	Counts the total number of rows in `orders`.

Note COUNT(*) includes all rows, while COUNT(column_name) excludes NULL values in the specified column.

Aggregate Functions

Clause | **How to use** | **Explained**

GROUP BY

```
  SELECT productLine, AVG(buyPrice)
         AS avg_price
    FROM products
GROUP BY productLine;
```

Groups rows by `productLine` and calculates the average price for each product line.

```
  SELECT productLine, AVG(buyPrice)
         AS avg_price
    FROM products
   WHERE buyPrice > 100
GROUP BY productLine;
```

Groups rows by `productLine` for products with price over 100 and calculates the average price for each product line.

```
  SELECT customerNumber, COUNT(orderNumber)
         AS order_count
    FROM orders
   WHERE orderDate >= '2023-01-01'
GROUP BY customerNumber
ORDER BY order_count DESC;
```

Groups orders by `customerNumber`, counts the number of orders for each customer in 2023 and beyond, and sorts the results by the order count in descending order.

This shows which customers placed the most orders in 2023.

Clause | **How to use** | **Explained**

HAVING

```
  SELECT productLine, AVG(buyPrice)
         AS avg_price
    FROM products
GROUP BY productLine
  HAVING AVG(buyPrice) > 50;
```

Filters product lines to only include those with average price greater than 50.

COUNT

```
SELECT COUNT(*) AS total_products
  FROM products;
```

Counts the total number of rows in the products table, returning the total number of products. This includes all rows, regardless of NULL values in any columns.

```
SELECT COUNT(reportsTo)
       AS employees_with_manager
  FROM employees;
```

Counts the number of non-null values in the reportsTo column of the employees table, showing how many employees have a manager assigned.

`COUNT` ignores `NULL` values, so employees without a manager (e.g., the president) are not included in this count.

String Functions

Clause | **How to use** | **Explained**

UPPER

```
SELECT UPPER(productName)
       AS uppercase_name
  FROM products;
```

Converts the `productName` column values to uppercase.

LOWER

```
SELECT LOWER(productName)
    AS lowercase_name
  FROM products;
```

Converts the `productName` column values to lowercase.

LENGTH

```
SELECT productName, LENGTH(productName)
    AS name_length
  FROM products;
```

Calculates the length of each value in the `productName` column.

SUBSTR

```
SELECT SUBSTR(productLine, 1, 3)
       AS product_category, productLine
  FROM products;
```

Extracts the first three characters from the `productLine` column. `SUBSTR` extracts a substring from a given string.

It can be used to extract characters from the beginning, end, or any position within the string.

```
SELECT SUBSTR(productCode, -4)
       AS product_id, productCode
  FROM products;
```

Extracts the last four characters from the `productCode` column.

CONCAT USING ||

```
SELECT firstName || ' ' || lastName
       AS full_name
  FROM employees;
```

Concatenates `firstName` and `lastName` with a space in between.

```
SELECT firstName || '.' || lastName ||
       '@classicmodelcars.com'
       AS email_address
  FROM employees;
```

Creates an email address by concatenating `firstName`, `lastName`, and domain.

Conditional Queries

Clause | **How to use** | **Explained**

CASE

```
SELECT productName,
      buyPrice,
      CASE
          WHEN buyPrice < 50 THEN
          'Budget'
          WHEN buyPrice BETWEEN 50
          AND 100 THEN 'Mid-range'
          ELSE 'Premium'
      END AS price_category
 FROM products;
```

Categorizes the `buyPrice` values into Budget, Mid-range, and Premium categories.

```
SELECT orderNumber,
       orderDate,
       CASE
           WHEN CAST(strftime('%m',
           orderDate) AS INTEGER)
           BETWEEN 3 AND 5 THEN
           'Spring Sale'

           WHEN CAST(strftime('%m',
           orderDate) AS INTEGER)
           BETWEEN 6 AND 8 THEN
           'Summer Sale'
           WHEN CAST(strftime('%m',
           orderDate) AS INTEGER)
           BETWEEN 9 AND 11 THEN
           'Fall Sale'
           ELSE 'Winter Sale'
       END AS sale_season
  FROM orders;
```

Categorizes orders into different sale seasons based on the order date.

COALESCE

```
SELECT productName,
       COALESCE(productDescription,
       'No description available')
       AS product_description
  FROM products;
```

Returns 'No description available' if `productDescription` is `null`.

```
SELECT employeeNumber,
       firstName,
       lastName,
       COALESCE(extension, email, 'No
       contact information') AS contact_info
  FROM employees;
```

Returns the first non-null value among extension, email, or 'No contact information'.

CAST

```
SELECT orderNumber, CAST(orderDate AS DATE)
       AS order_day
  FROM orders;
```

Converts the `orderDate` to `DATE` type.

Combine Data

Clause	How to use	Explained
`UNION`	`SELECT productName FROM products WHERE productLine = 'Classic Cars' UNION SELECT productName FROM products WHERE productLine = 'Vintage Cars';`	Combines the product names from 'Classic Cars' and 'Vintage Cars' product lines, removing duplicates.
`UNION ALL`	`SELECT productName FROM products WHERE productLine = 'Classic Cars' UNION ALL SELECT productName FROM products WHERE productLine = 'Vintage Cars';`	Combines the product names from 'Classic Cars' and 'Vintage Cars' product lines without removing duplicates.
`EXCEPT`	`SELECT productCode, productName FROM products EXCEPT SELECT productCode, productName FROM products WHERE productLine = 'Classic Cars';`	Returns products `EXCEPT` the 'Classic Cars' product line, demonstrating how `EXCEPT` removes rows from the first result that appear in the second result.
`INTERSECT`	`SELECT customerNumber, customerName FROM customers WHERE country = 'USA' INTERSECT SELECT customerNumber, customerName FROM customers WHERE creditLimit > 100000;`	Returns customers who are both located in the USA and have a credit limit over 100,000. This query demonstrates how `INTERSECT` finds common rows between two result sets.

Code as printed:

```sql
SELECT productName
  FROM products
 WHERE productLine = 'Classic Cars'
 UNION
SELECT productName
  FROM products
 WHERE productLine = 'Vintage Cars';
```

```sql
SELECT productName
  FROM products
 WHERE productLine = 'Classic Cars'
 UNION ALL
SELECT productName
  FROM products
 WHERE productLine = 'Vintage Cars';
```

```sql
SELECT productCode, productName
  FROM products
EXCEPT
SELECT productCode, productName
  FROM products
 WHERE productLine = 'Classic Cars';
```

```sql
SELECT customerNumber, customerName
  FROM customers
 WHERE country = 'USA'
INTERSECT
SELECT customerNumber, customerName
  FROM customers
 WHERE creditLimit > 100000;
```

Note EXCEPT and INTERSECT are not supported in all SQL databases. These examples use PostgreSQL syntax.

Window Functions

Note SQLite does not support window functions natively. The following examples use PostgreSQL syntax and require PostgreSQL or a SQLite extension.

Clause | **How to use** | **Explained**

`PARTITION BY`

```
SELECT employeeNumber,
       officeCode,
       extension,
       AVG(LENGTH(extension)) OVER (
           PARTITION BY officeCode
       ) AS avg_extension_length
  FROM employees;
```

Calculates the average extension length within each office. The `PARTITION BY` clause divides the data into partitions based on the `officeCode` column.

`ORDER BY`

```
SELECT employeeNumber,
       officeCode,
       extension,
       SUM(LENGTH(extension)) OVER (
           ORDER BY LENGTH(extension) DESC
      ) AS running_total_length
  FROM employees;
```

Calculates a running total of extension lengths ordered by length in descending order.

`PARTITION BY`

`ORDER BY`

```
SELECT employeeNumber,
       officeCode,
       extension,
       SUM(LENGTH(extension)) OVER (
           PARTITION BY officeCode
           ORDER BY LENGTH(extension) DESC
       ) AS running_total_length
  FROM employees;
```

Calculates a running total of extension lengths within each office, ordered by length.

Ranking Functions

Note SQLite does not support ranking functions natively. The following examples use PostgreSQL syntax and require PostgreSQL or a SQLite extension.

Clause | **How to use** | **Explained**

`DENSE RANK`

```
SELECT productCode,
       productName,
       buyPrice,
       DENSE_RANK() OVER (
           ORDER BY buyPrice DESC
       ) AS price_rank
  FROM products;
```

Ranks products based on `buyPrice` in descending order. Differs from `RANK` by handling ties differently (no gaps in ranking).

`RANK`

```
SELECT employeeNumber,
       officeCode,
       extension,
       RANK() OVER (
           PARTITION BY officeCode
           ORDER BY LENGTH(extension) DESC
       ) AS extension_rank_in_office
  FROM employees;
```

Ranks employees within each office based on their extension length. Differs from `DENSE_RANK` by leaving gaps in ranking when there are ties.

`ROW NUMBER`

```
SELECT orderNumber,
       orderDate,
       customerNumber,
       ROW_NUMBER() OVER (
           ORDER BY orderDate,
                    customerNumber
       ) AS order_number
  FROM orders;
```

Assigns a unique row number to each order based on `orderDate` and `customerNumber`.

Joins

Clause | **How to use** | **Explained**

INNER JOIN

```sql
SELECT o.orderNumber,
       o.orderDate,
       c.customerName
  FROM orders AS o
 INNER JOIN customers AS c
    ON o.customerNumber = c.customerNumber;
```

Joins `orders` and `customers` tables, returning only matching rows. This is the default join type when `JOIN` is used without specifying `LEFT`, `RIGHT`, or `FULL`.

LEFT JOIN

```sql
SELECT p.productCode,
       p.productName,
       od.orderNumber
  FROM products AS p
  LEFT JOIN orderdetails AS od
    ON p.productCode = od.productCode;
```

Joins `products` and `orderdetails` tables, returning all products and their orders (if any).

RIGHT JOIN

```sql
SELECT e.employeeNumber,
       e.lastName,
       o.officeCode
  FROM offices AS o
 RIGHT JOIN employees AS e
    ON o.officeCode = e.officeCode;
```

Joins `offices` and `employees` tables, returning all employees and their offices (if any).

Clause | **How to use** | **Explained**

CROSS JOIN

```sql
SELECT p.productName,
       pl.textDescription
  FROM products AS p
 CROSS JOIN productlines AS pl;
```

Returns all possible combinations of products and product line descriptions.

JOIN MULTIPLE

```sql
SELECT o.orderNumber,
       c.customerName,
       p.productName
  FROM orders AS o
  JOIN customers AS c
    ON o.customerNumber = c.customerNumber
  JOIN orderdetails AS od
    ON o.orderNumber = od.orderNumber
  JOIN products p
    ON od.productCode = p.productCode;
```

Joins four tables: `orders`, `customers`, `orderdetails`, and `products`.

JOIN SELF

```sql
SELECT e1.firstName || ' ' || e1.lastName
AS employee,
       e2.firstName || ' ' || e2.lastName
AS manager
  FROM employees AS e1
  LEFT JOIN employees AS e2
    ON e1.reportsTo = e2.employeeNumber;
```

Self-join example listing employees and their respective managers.

Subqueries

Clause | How to use | Explained

SUBQUERY IN SELECT

```sql
SELECT productName,
       buyPrice,
       (SELECT AVG(buyPrice) FROM
       products) AS avg_price
  FROM products;
```

Includes a subquery that calculates the average price for all products.

SUBQUERY IN FROM

```sql
SELECT productLine,
       avg_price
  FROM (SELECT productLine,
               AVG(buyPrice) AS avg_price
          FROM products
         GROUP BY productLine)
         AS line_averages
 WHERE avg_price > 100;
```

Finds product lines with an average price greater than 100 using a subquery.

SUBQUERY IN WHERE

```sql
SELECT productName,
       buyPrice
  FROM products p1
 WHERE p1.buyPrice > (
       SELECT AVG(p2.buyPrice)
         FROM products p2
        WHERE p1.productLine =
              p2.productLine)
 ORDER BY productLine,
          buyPrice DESC;
```

This query selects products that are more expensive than the average price in their respective product line, ordered by product line and price in descending order.

Clause | How to use | Explained

SUBQUERY WITH IN

```sql
SELECT productName,
       buyPrice
  FROM products
 WHERE productCode IN (
       SELECT productCode
         FROM orderdetails
        WHERE orderNumber = 10100
       );
```

Finds products that were ordered in order 10100.

SUBQUERY WITH EXISTS

```sql
SELECT customerName
  FROM customers c
 WHERE EXISTS (
       SELECT 1
         FROM orders o
        WHERE o.customerNumber
              = c.customerNumber
          AND o.orderDate >= '2023-01-01'
       );
```

This query retrieves the names of customers who have placed at least one order on or after January 1, 2023.

Subqueries

Clause	How to use	Explained
=	(see code below)	This query selects all orders for a specific customer named 'Mini Gifts Distributors Ltd.', ordered by date from most recent to oldest.
CTE	(see code below)	This query calculates the total amount for each order using a `CTE` and then joins the `orders` table with the `CTE` to display order details with total amounts, ordered by total amount in descending order.

=

```sql
SELECT orderNumber,
       orderDate,
       totalAmount
  FROM orders
 WHERE customerNumber = (
       SELECT customerNumber
         FROM customers
        WHERE customerName = 'Mini Gifts
              Distributors Ltd.'
       )
 ORDER BY orderDate DESC;
```

This query selects all orders for a specific customer named 'Mini Gifts Distributors Ltd.', ordered by date from most recent to oldest.

CTE

```sql
WITH order_totals AS (
     SELECT orderNumber,
            SUM(quantityOrdered * priceEach)
            AS total_amount
       FROM orderdetails
      GROUP BY orderNumber
)
SELECT o.orderNumber,
       o.orderDate,
       ot.total_amount
  FROM orders o
  JOIN order_totals ot
       ON o.orderNumber = ot.orderNumber
 ORDER BY ot.total_amount DESC;
```

This query calculates the total amount for each order using a `CTE` and then joins the `orders` table with the `CTE` to display order details with total amounts, ordered by total amount in descending order.

SQLite and PostgreSQL

SQLite Commands

`.tables`	Lists all tables in the current database.
`.schema table_name`	Shows the schema for the specified table.

`.mode column`
`.headers on`
	Sets output to column mode with headers for better readability.

`.open filename`	Opens a new or existing database file.
`.save filename`	Saves the current database to a file.
`.quit`	Exits the SQLite prompt.

SQLite and PostgreSQL

PostgreSQL Commands

`\l`	Lists all databases.
`\c database_name`	Connects to a specific database.
`\dt`	Lists all tables in the current database.
`\d table_name`	Describes the specified table.
`\du`	Lists all roles/users.
`\timing`	Toggles display of query execution time.
`\e`	Opens the last command in an editor.
`\i filename`	Executes commands from a file.
`\q`	Exits the PostgreSQL interactive terminal.

Note SQLite doesn't have a built-in user management system like PostgreSQL, so commands related to user management are not applicable.