Advanced-Intermediate SQL

Chapter 11

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Introduction

Motivation

With the SQL tools we have learned so far, we can answer many standard data analysis questions by leveraging the relationships in relational databases.

However, for more complex analytical tasks, relying solely on joins or nested subqueries can lead to queries that are difficult to write, read, and debug.

We need additional tools that allow us that help us structure our logic more clearly and often reuse it more effectively.

Advanced-Intermediate SQL

To round out our core SQL toolkit for data analytics and data science, we will introduce three powerful concepts that go beyond basic joins and subqueries:

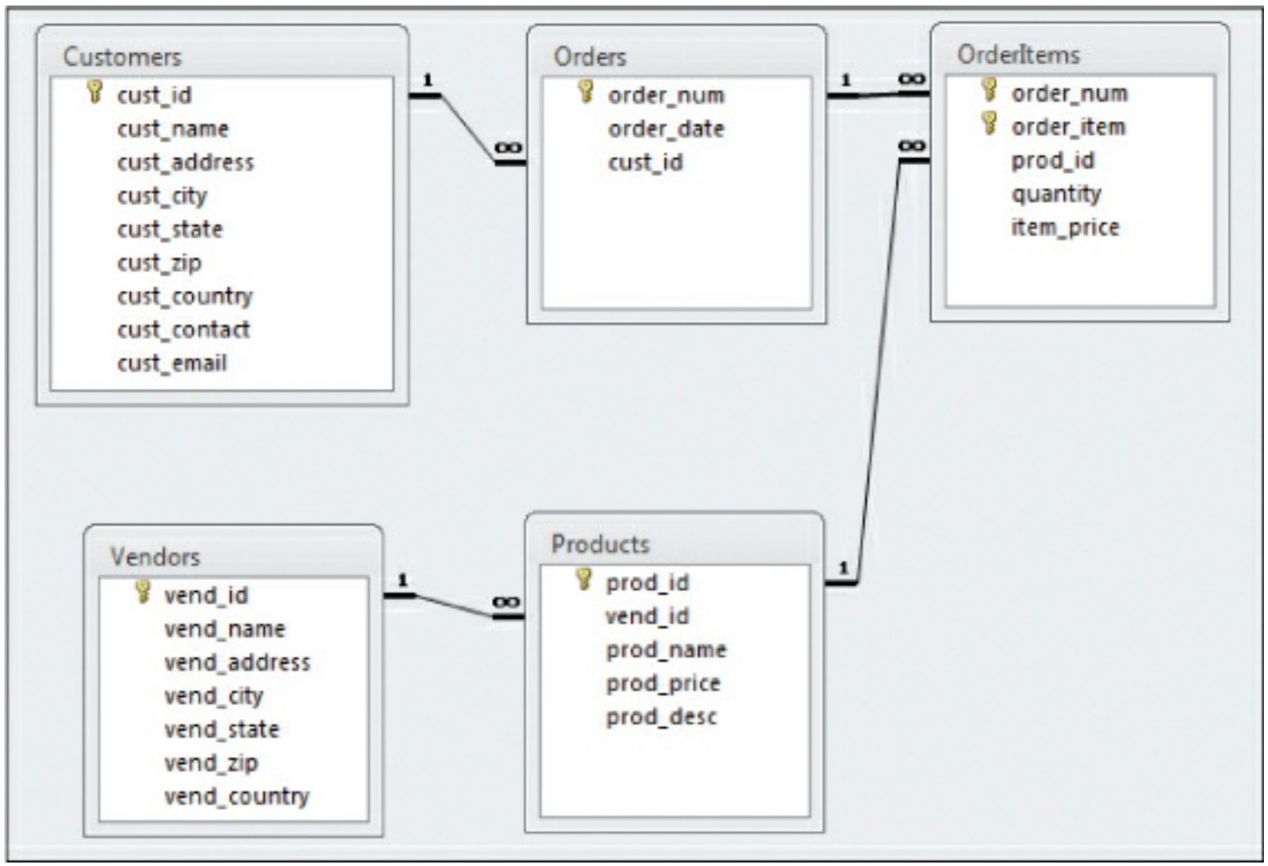
- Window functions
- Common table expressions (CTEs)
- Views

These tools will allow us to complete more advanced data tasks with cleaner and more maintainable SQL code.

In particular, with this core advanced-intermediate SQL toolkit, you will be able to solve nearly any SQL question you might encounter in an entry-level data science interview (including those on platforms like Leetcode or Stratascratch).

Schema of the TYSQL Data

We will continue to use the TYSQL database, which contains data for a fictitious toy store. The schema of the TYSQL database:



Source: Ben Forta, *SQL in 10 Minutes a Day, Sams Teach Yourself, 5th Edition*, Pearson, 2020.

Window Functions

Motivating Example

Suppose we want a list of the line items (i.e., distinct product IDs) from each order that also shows the total number of line items within each order.

For example, for order numbers 20005 and 20006, we want:

order_num	prod_id	quantity	num_its
20005	BR01	100	2
20005	BR03	100	2
20006	BR01	20	3
20006	BR02	10	3
20006	BR03	10	3

How can we write a SQL query that returns this result set (for all orders)?

Hint: What are the steps needed to output this result?

Step 1: GROUP BY

The first step is to find the number of (line) items in each order.

We can use a `GROUP BY` query with the `COUNT()` aggregate function:

```
SELECT order_num, COUNT(*) AS num_its
FROM OrderItems
GROUP BY order_num;
```

order_num	num_its
20005	2
20006	3
20007	5
20008	5
20009	3

Note that this query result has aggregated the rows of the `OrderItems` table to show a single count value for each order. It does not show the individual rows of `OrderItems`.

How can we use these counts to now find the result we want?

Step 2: Join on a Derived Table

Next, we need to join the original `OrderItems` table with the `num_its` counts.

The preceding query result can be used as a **derived table**, i.e., a subquery using the result as if it were a table.

```
SELECT oi.order_num, prod_id, quantity, num_its
FROM OrderItems AS oi
INNER JOIN (
   SELECT order_num, COUNT(*) AS num_its
   FROM OrderItems
   GROUP BY order_num
) AS counts
ON oi.order_num = counts.order_num;
```

Notice that we need an alias for the derived table to specify the join condition.

Full Subquery with Output

```
SELECT oi.order_num, prod_id, quantity, num_its
FROM OrderItems AS oi
INNER JOIN (
   SELECT order_num, COUNT(*) AS num_its
   FROM OrderItems
   GROUP BY order_num
) AS counts
ON oi.order_num = counts.order_num;
```

order_num	prod_id	quantity	num_its
20005	BR01	100	2
20005	BR03	100	2
20006	BR01	20	3
20006	BR02	10	3
20006	BR03	10	3
20007	BR03	50	5
20007	BNBG01	100	5
20007	BNBG02	100	5
20007	BNBG03	100	5
20007	RGAN01	50	5

order_num	prod_id	quantity	num_its
20008	RGAN01	5	5
20008	BR03	5	5
20008	BNBG01	10	5
20008	BNBG02	10	5
20008	BNBG03	10	5
20009	BNBG01	250	3
20009	BNBG02	250	3
20009	BNBG03	250	3

Moving Beyond Group By

The subquery solution to add group-level information to each row is logically straightforward in this example:

1. Compute the group summary using an aggregate function.
2. Join the summary back using a derived table.

However, for more complex queries or more complicated group-level computations, this approach can become repetitive, harder to read, and computationally inefficient.

For example:

- What if we wanted to find the running total number of units of each product sold over time?
- What if we want to order each line item within an order by the quantity purchased?

Is there a better way to perform within-group computations while still retaining each individual row?

Window Functions

A **window function** is a function that is applied to each row of a result set using a defined set of related rows called a **window**.

While logically similar to aggregate functions used with `GROUP BY`, a window function does not collapse rows into groups. Instead, it retains individual rows and computed values are added to each one.

Source: https://www.sqlitetutorial.net/sqlite-window-functions/

Window Function Syntax

The basic syntax for window functions is shown as follows:

```
window_function() OVER (
   PARTITION BY column_name
   ORDER BY column_name
   ROWS or RANGE or GROUP window_frame_extent
)
```

- The **`OVER`** clause defines the windows over which the window function will be applied. This is the only mandatory clause.
- The `PARTITION BY` clause specifies how rows are divided into groups (or **partitions**). The window function is applied independently within each partition.
- The `ORDER BY` clause specifies the order in which rows are processed within each partition.
- Window functions are computed row by row. A **window frame** is the subset of rows within a window that the function will compute over based on their position relative to the current row. If the windows are ordered (i.e., if `ORDER BY` is specified), the `window_frame_extent` specifies the rows or range of the window frame.

Example: Number of Line Items (Window Function)

Revisiting our earlier example, we want a list of the line items from each order that also shows the total number of line items within each order.

```
SELECT order_num,
       prod_id,
       quantity,
       COUNT(*) OVER (
         PARTITION BY order_num
       ) AS num_its
FROM OrderItems;
```

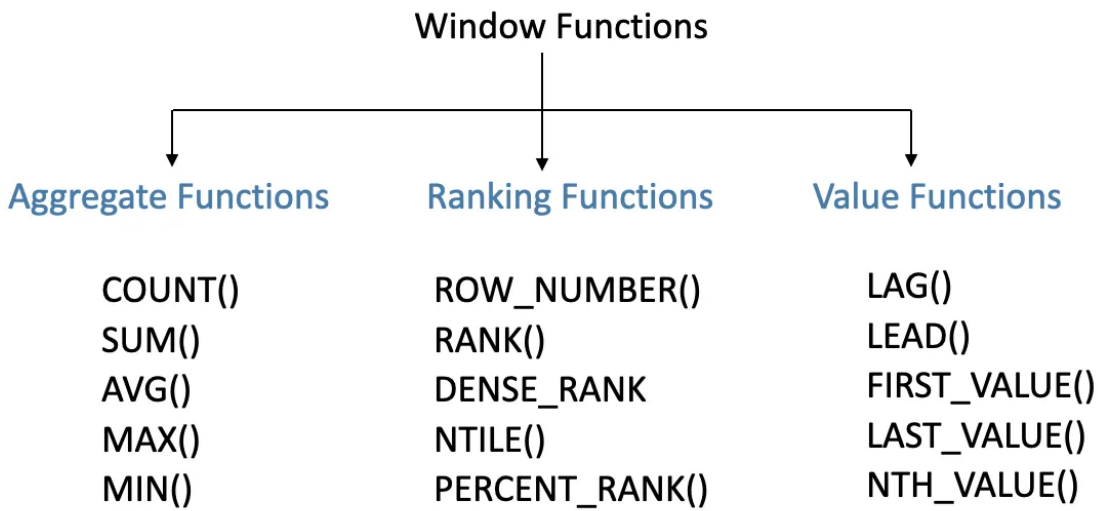
order_num	prod_id	quantity	num_its
20005	BR01	100	2
20005	BR03	100	2
20006	BR01	20	3
20006	BR02	10	3
20006	BR03	10	3
20007	BR03	50	5
20007	BNBG01	100	5
20007	BNBG02	100	5
20007	BNBG03	100	5
20007	RGAN01	50	5
20008	RGAN01	5	5
20008	BR03	5	5
20008	BNBG01	10	5
20008	BNBG02	10	5
20008	BNBG03	10	5
20009	BNBG01	250	3
20009	BNBG02	250	3
20009	BNBG03	250	3

Window Function Categories

Window functions are often divided into three categories:

- Aggregate window functions
- Ranking window functions
- Value window functions

An overview of the window function categories is shown below:



Source: Medium article "The Power of Window Functions in SQL" by AnalystHub, 2023.

Aggregate Window Functions

The usual aggregate functions used with `GROUP BY` are available as window functions, as they are just applied over windows.

Function	Description
`COUNT(expr) OVER (window)`	Find the number of rows with a non-null expression in the window
`SUM(expr) OVER (window)`	Finds the sum of the expression in the window
`AVG(expr) OVER (window)`	Find the average (mean) of the expression in the window
`MAX(expr) OVER (window)`	Finds the maximum of the expression in the window
`MIN(expr) OVER (window)`	Finds the minimum of the expression in the window

By including an `ORDER BY` clause in the `window` expression, the functions will compute running (cumulative) values.

Example: Running Total

A very common application of window functions is to compute running (or cumulative) totals. A running total requires ordering the rows within the window.

```
SELECT order_num,
       prod_id,
       quantity,
       SUM(quantity) OVER (
         PARTITION BY order_num
         ORDER BY prod_id
       ) AS run_tot
FROM OrderItems;
```

order_num	prod_id	quantity	run_tot
20005	BR01	100	100
20005	BR03	100	200
20006	BR01	20	20
20006	BR02	10	30
20006	BR03	10	40
20007	BNBG01	100	100
20007	BNBG02	100	200
20007	BNBG03	100	300
20007	BR03	50	350
20007	RGAN01	50	400

order_num	prod_id	quantity	run_tot
20008	BNBG01	10	10
20008	BNBG02	10	20
20008	BNBG03	10	30
20008	BR03	5	35
20008	RGAN01	5	40
20009	BNBG01	250	250
20009	BNBG02	250	500
20009	BNBG03	250	750

Example: Running Totals Over Time

Combining a join with a window function, we can find the running total number of units of each product sold over time.

```
SELECT order_date, prod_id,
       SUM(quantity) OVER (
         PARTITION BY prod_id
         ORDER BY order_date
       ) AS cume_ord
FROM OrderItems, Orders
WHERE OrderItems.order_num = Orders.order_num;
```

order_date	prod_id	cume_ord
2020-01-30	BNBG01	100
2020-02-03	BNBG01	110
2020-02-08	BNBG01	360
2020-01-30	BNBG02	100
2020-02-03	BNBG02	110
2020-02-08	BNBG02	360
2020-01-30	BNBG03	100
2020-02-03	BNBG03	110
2020-02-08	BNBG03	360

order_date	prod_id	cume_ord
2020-01-12	BR01	20
2020-05-01	BR01	120
2020-01-12	BR02	10
2020-01-12	BR03	10
2020-01-30	BR03	60
2020-02-03	BR03	65
2020-05-01	BR03	165
2020-01-30	RGAN01	50
2020-02-03	RGAN01	55

Ranking Window Functions

Ranking window functions assign rankings to rows within windows. Each of these functions requires an `ORDER BY` sub-clause within the `OVER` clause.

Function	Description
`ROW_NUMBER()`	Assigns a sequential integer to each row within the partition
`RANK()`	Assigns a rank number to each row (skips ranks when there are ties)
`DENSE_RANK()`	Like `RANK()` but does not skip ranks when there are ties
`PERCENT_RANK()`	Like `RANK()` but assigns the rank as a percentage
`NTILE(n)`	Assigns rows into `n` roughly equal-sized buckets (or tiles)
`CUME_DIST()`	The percentage of rows less than or equal to the current row (i.e., the cumulative distribution)

Example: Ranks and Cumulative Proportions of Orders

For example, we can find the ranks based on order number and the cumulative proportion of orders over time:

```
SELECT order_num,
       order_date,
       CUME_DIST() OVER (
         ORDER BY order_date
       ) AS cume_prop,
       RANK() OVER (
         ORDER BY order_num
       ) AS order_rank
FROM Orders;
```

order_num	order_date	cume_prop	order_rank
20006	2020-01-12	0.2	2
20007	2020-01-30	0.4	3
20008	2020-02-03	0.6	4
20009	2020-02-08	0.8	5
20005	2020-05-01	1.0	1

Value Window Functions

Value window functions return specific values from a row within the window. Each of these functions requires an `ORDER BY` sub-clause within the `OVER` clause.

Function	Description
`LAG(column, offset)`	Returns a value from a previous row (offset defaults to 1)
`LEAD(column, offset)`	Returns a value from a later row (offset defaults to 1)
`FIRST_VALUE(column)`	Returns the first value in an ordered set of values
`LAST_VALUE(column)`	Returns the last value in an ordered set of values
`NTH_VALUE(column, n)`	Returns the `n`th value in an ordered set of values

Specifying the Window Frame

Many of the window functions are affected by the window frame, which specifies the context of the computations as they are done row by row.

The window frame is defined according to one of the following syntax rules:

```
ROWS or RANGE or GROUP frame_start
ROWS or RANGE or GROUP BETWEEN frame_start AND frame_end
```

- `ROWS` defines the frame by the row positions relative to the current row.
- `RANGE` defines the frame by the values in the `ORDER BY` column(s).
- `GROUP` defines the frame by peer groups sharing the same `ORDER BY` value.

The `frame_start` is one of the following options:

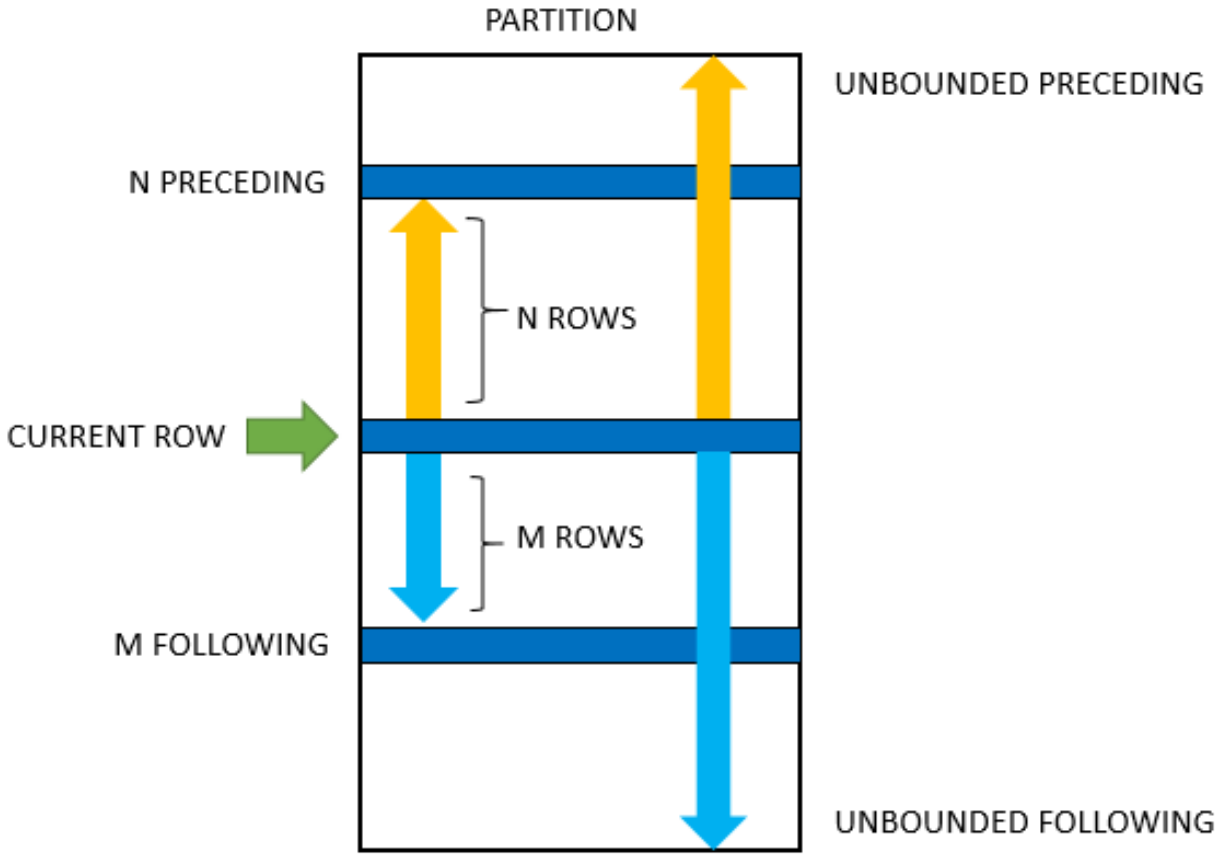
- `N PRECEDING`
- `UNBOUNDED PRECEDING`
- `CURRENT ROW`

The `frame_end` is one of the following options:

- `CURRENT ROW`
- `UNBOUNDED FOLLOWING`
- `N FOLLOWING`

Window Frame Illustration

An illustration of the window frame and its options:



Source: Imply.io article "An Introduction to Window Functions" by Kumar Abhishek.

Default Window Frames

The default window frame depends on whether there is an `ORDER BY` clause in the window definition.

If `ORDER BY` is specified, the default window frame is:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

This makes the window function behave like a running total.

If `ORDER BY` is not specified, the default window frame is:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

This window frame includes all rows in the partition.

Named Windows: The WINDOW Clause

If a window is used for several window functions in the same query, it can be repetitive to define the same window for each window function.

To reuse the same window for multiple window functions, the `WINDOW` clause can define a **named window** that can then be referenced elsewhere in the query.

The syntax to define a named window is as follows:

```
SELECT window_function() OVER(window_name)
FROM ...
[HAVING ...]
WINDOW window_name AS (
   PARTITION BY column_name
   ORDER BY column_name
   ROWS or RANGE or GROUP window_frame_extent
)
[ORDER BY ...]
```

Note that the `WINDOW` clause must be between the `HAVING` and `ORDER BY` clauses (if used).

Example: Named Window

```
SELECT order_num,
       prod_id,
       quantity,
       SUM(quantity) OVER (win) AS run_tot,
       COUNT(*) OVER (win) AS run_count
FROM OrderItems
WINDOW win AS (
   PARTITION BY order_num
   ORDER BY prod_id
)
LIMIT 10;
```

order_num	prod_id	quantity	run_tot	run_count
20005	BR01	100	100	1
20005	BR03	100	200	2
20006	BR01	20	20	1
20006	BR02	10	30	2
20006	BR03	10	40	3
20007	BNBG01	100	100	1
20007	BNBG02	100	200	2
20007	BNBG03	100	300	3
20007	BR03	50	350	4
20007	RGAN01	50	400	5

Common Table Expressions (CTEs)

Motivating Example

Let's revisit the earlier example:

We wanted to find a list of the line items from each order that also shows the total number of line items within each order.

We first found a solution using a subquery (derived table) and a join:

```
SELECT oi.order_num, prod_id, quantity, num_its
FROM OrderItems AS oi
INNER JOIN (
   SELECT order_num, COUNT(*) AS num_its
   FROM OrderItems
   GROUP BY order_num
) AS counts
ON oi.order_num = counts.order_num;
```

This solution translates the logic of the two-step solution (compute the group summary, join the summary back), but it does so in a way that can be difficult to read because the summary step is nested inside the join step.

Is there a way to break up complex queries into more readable parts that flow more logically?

Common Table Expressions (CTEs)

A **common table expression** (or **CTE**) is a named temporary result set that can be referenced within the scope of a single, larger SQL statement.

A CTE can be thought of intuitively like a named subquery.

The basic syntax for defining a CTE is as follows:

```
WITH cte_name (col1, col2, ...) AS (
    SELECT ...
)
```

- The `WITH` keyword is used to define the CTE.
- The name `cte_name` can be referenced in the main SQL statement.
- The list of column names `(col1, col2, ...)` is optional but can be used to name columns of the CTE's result set.

Example: Number of Line Items (CTE)

The earlier subquery and join can be written with a CTE:

```
WITH counts AS (
    SELECT order_num, COUNT(*) AS num_its
    FROM OrderItems
    GROUP BY order_num
)
SELECT oi.order_num, prod_id, quantity, counts.num_its
FROM OrderItems AS oi
INNER JOIN counts
ON oi.order_num = counts.order_num;
```

order_num	prod_id	quantity	num_its
20005	BR01	100	2
20005	BR03	100	2
20006	BR01	20	3
20006	BR02	10	3
20006	BR03	10	3
20007	BR03	50	5
20007	BNBG01	100	5
20007	BNBG02	100	5
20007	BNBG03	100	5

order_num	prod_id	quantity	num_its
20007	RGAN01	50	5
20008	RGAN01	5	5
20008	BR03	5	5
20008	BNBG01	10	5
20008	BNBG02	10	5
20008	BNBG03	10	5
20009	BNBG01	250	3
20009	BNBG02	250	3
20009	BNBG03	250	3

Usefulness of CTEs

CTEs have a variety of uses:

- CTEs can break up complicated SQL statements into smaller, more manageable parts. This helps in writing SQL code that is more logically written and easier to read, write, and debug.
- If the result set from a subquery is used multiple times in a query, a CTE is useful to avoid writing redundant subqueries.
- Logic can be encapsulated by chaining together multiple CTEs within a single `WITH` clause, with each CTE separated by a comma. For complex logic, subsequent CTEs can also reference earlier ones.
- **Recursive CTEs** (uses the keywords `WITH RECURSIVE`) are a special type of CTE that references itself within its own definition. These can be used to write recursive queries for hierarchical or tree-structured data (e.g., organizational charts, directories, threaded discussions).

Note: A CTE is temporary and only defined for a single SQL statement. A single CTE definition cannot be used for multiple statements (there are other constructs for that).

Example: Percentage of Revenue

Suppose we want to see the percentage of revenue each product has generated across all orders.

We want a list of all products that have been ordered, where for each product, we show the number of units sold, the total revenue from each product, the total revenue across all products, and the product's share of the total revenue across all products.

The result set we want:

prod_id	product_revenue	total_revenue	revenue_pct
BNBG01	956.40	5730.7	16.69
BNBG02	956.40	5730.7	16.69
BNBG03	956.40	5730.7	16.69
BR01	668.80	5730.7	11.67
BR02	89.90	5730.7	1.57
BR03	1853.35	5730.7	32.34
RGAN01	249.45	5730.7	4.35

Example: Percentage of Revenue (Derived Tables)

A solution using derived tables:

```
SELECT prod_id,
       product_revenue,
       total_revenue,
       ROUND(100 * product_revenue / total_revenue, 2)
         AS revenue_pct
FROM (
   SELECT prod_id,
          SUM(quantity * item_price) AS product_revenue
   FROM OrderItems
   GROUP BY prod_id
) AS per_product,
(
   SELECT SUM(quantity * item_price) AS total_revenue
   FROM OrderItems
) AS total;
```

Example: Percentage of Revenue (Multiple CTEs)

A solution using multiple CTEs:

```
WITH ProductRevenue AS (
    SELECT prod_id,
           SUM(quantity * item_price) AS product_revenue
    FROM OrderItems
    GROUP BY prod_id
),
TotalRevenue AS (
    SELECT SUM(quantity * item_price) AS total_revenue
    FROM OrderItems
)
SELECT
  pr.prod_id,
  pr.product_revenue,
  tr.total_revenue,
  ROUND(100 * pr.product_revenue / tr.total_revenue, 2)
   AS revenue_pct
FROM ProductRevenue AS pr, TotalRevenue AS tr;
```

When To Use Subqueries Over CTEs

CTEs are like named subqueries, so it may be tempting to rewrite *every* subquery as a CTE. But in some cases, traditional subqueries are still the preferred style.

- For short and/or single-use subqueries, an in-line subquery may be more concise, such as when a single-valued or short subquery is used inside a `WHERE` clause.
- Since CTEs are written before the main statement, the logic of a CTE is not necessarily near where the result is used in the main statement. For straightforward statements, subqueries can be more readable.

Views

Reusable Named Queries

CTEs are powerful tools for encapsulating logic by defining named queries that can be used and reused within a single SQL statement.

However, a CTE only exists for the duration of that one statement.

What if we want to reuse a named query across multiple SQL statements?

Views

A **view** is a saved query that acts like a virtual table.

The syntax for creating a view is as follows:

```
CREATE [TEMP | TEMPORARY] VIEW [IF NOT EXISTS] view_name AS
-- Query goes here
```

Unlike CTEs, a view (by default) persists in the database until it is explicitly dropped using `DROP VIEW view_name`.

Including the optional `TEMP` or `TEMPORARY` keyword will create a **temporary view**, which only persists for a session. A temporary view will only be visible to the database connection that created it and is automatically deleted when the database connection is closed.

Including the optional `IF NOT EXISTS` keywords will only create the view if a view of the same name does not already exist.

Note: In SQLite, all views are read-only, so they cannot be changed with `DELETE`, `INSERT`, or `UPDATE` (like they can in other dialects). To update a view in SQLite, the view can be dropped and then recreated.

Views Are Saved Queries Not Saved Results

Views can be queried, joined, or filtered just like regular tables, but they *do not store data*. Instead, a view dynamically retrieves data by executing its underlying query each time it is accessed.

Since views save queries and not results, they always reflect the most current data in the database: If the underlying tables change, the view's results automatically reflect those updates.

Example: TYSQL Views

The `TYSQL` database already has two views that represent common joins we might need: `CustomersWithOrders` and `ProductCustomers`.

We can see the query that created the view in the database metadata:

```
SELECT sql
FROM sqlite_master
WHERE tbl_name = 'ProductCustomers';
```

```
CREATE VIEW ProductCustomers AS
SELECT cust_name, cust_contact, prod_id
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num
```

Example: ProductCustomers

To find the information for customers who have ordered products, we can use the `ProductCustomers` view directly:

```
SELECT DISTINCT cust_name, cust_contact
FROM ProductCustomers;
```

cust_name	cust_contact
Village Toys	John Smith
Fun4All	Jim Jones
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

Usefulness of Views

Views are particularly useful for abstraction (i.e., hiding complexity) and data security.

Abstraction

- Much like CTEs, views are useful for encapsulating complicated joins and conditions into a persistent virtual table.

  For commonly reused logic across multiple queries, creating a view can reduce writing long and/or redundant statements.

Data security

- For many databases, users have different privileges that restrict access to certain tables in the data.

  To prevent direct access or modification of base tables (that could contain sensitive information), users can be granted access only to a filtered view, not to the underlying tables.

Temporary Tables

A **temporary table** is like a real table in that it *stores data* on disk, but it only persists for a session.

The syntax for creating a temporary table is as follows:

```
CREATE [TEMP | TEMPORARY] TABLE AS
-- Query goes here
```

A temporary table is like a real table in that it *stores data* on disk, but it only persists for a session.

A temporary view, on the other hand, is a saved query for a session. It does not store data, as it will execute its query each time it is accessed.

In essence, a view (either temporary or persistent) saves the *logic* of the query, while a temporary table saves the *result* of the query.

Since temporary tables do not need to retrieve data every time they are used, they can be much faster than views if reused many times.