CSC 365

Introduction to Database Systems

Common Table Expressions (CTE) - WITH

`WITH` allows you to define temporary result sets (tables) that exist for a single query. These auxiliary queries are referred to as a Common Table Expressions (CTEs).

```
WITH <CTE1 name> AS (
  SELECT ...
)[, <CTE2 name> AS (
  SELECT ...
)]
SELECT * FROM <table>, <CTE1 name>, <CTE2 name> ...
```

CTE / WITH Example

CAL POLY

```
WITH monthly_sales AS (
  SELECT monthname(TransDate) AS Month, SUM(Amount) AS Sales
  FROM accounting_entries
  WHERE AccountType = 'Sales'
  GROUP BY monthname(TransDate)
),
monthly_cogs AS (                                    -- COGS: Cost of Goods Sold
  SELECT monthname(TransDate) AS Month, SUM(Amount) AS COGS
  FROM accounting_entries
  WHERE AccountType = 'COGS'
  GROUP BY monthname(TransDate)
)
SELECT T.AccountType, T.Account,
  TransDate, monthname(TransDate) as Month,
  Amount,
  C.COGS as MonthCOGSTotal,
  S.Sales as MonthSalesTotal,
  ROUND((T.Amount / S.Sales) * 100, 2) AS PctOfMonthlySales
FROM accounting_entries T
  INNER JOIN monthly_sales S ON monthname(TransDate) = S.Month
  INNER JOIN monthly_cogs C ON monthname(TransDate) = C.Month
ORDER BY TransDate, AccountType
```

CTE / WITH Example

Find the most popular flavor(s) of Eclair sold at the BAKERY, based on number of purchases.

Among the possible approaches:

1. Nesting
2. CTE (eliminate some redundancy)

CTE Considerations

CAL POLY

- Can simplify certain queries, avoid redundancy
- Just like Window Functions, supported only in the latest version of MySQL
  - Good support in other popular RDBMSs
- Optimization "fence" in *some* cases
  - CTE query might be materialized as table, without any algebraic laws applied across the entire query
  - Consult DBMS documentation and query plan

WITH RECURSIVE

`WITH RECURSIVE` allows a `WITH` query to refer to its *own* output. To sum the numbers from 1 to 100:

```
WITH RECURSIVE t(n) AS (
   SELECT 1
   UNION ALL
   SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

See Also:

- Solving the Traveling Salesman Problem with Postgres Recursive CTEs
- SQL 3D Engine

WITH RECURSIVE - Flights

CAL POLY

```
-- Enumerate all flights from airport AMW with <= 3 transfers
WITH RECURSIVE routes(the_source, the_dest, route, hops) as (
  SELECT Source as the_source, Destination,
  CAST(CONCAT(Source, '->', Destination) AS CHAR(150)) as route, 0 as hops
  FROM AIRLINES.flights
  WHERE Source = 'AMW'
  UNION
  SELECT routes.the_source, f.Destination,
  CONCAT(route, '->', f.Destination),
  routes.hops + 1 as hops
  FROM AIRLINES.flights f, routes
  WHERE f.Source = routes.the_dest and f.Destination <> 'AMW' AND hops < 3
)
SELECT * FROM routes
```

Common Table Expressions (CTEs)

- Common Table Expressions ()
  - `WITH`
  - `WITH RECURSIVE`

Introduced with the SQL 2005 standard, added to MySQL in version 8.0, released several years ago

Common Table Expressions (CTE) - WITH

CAL POLY

`WITH` allows you to define temporary result sets (tables) that exist for a single query. These auxiliary queries are referred to as a Common Table Expressions (CTEs).

```
WITH <CTE1 name> AS (
  SELECT ...
), <CTE2 name> AS (
  SELECT ...
)
SELECT * FROM <table>, <CTE1 name>, <CTE2 name> ...
```

CTE / WITH Example (PostgreSQL)

CAL POLY

```
with small_flightnos as (
  select FlightNo, Source, Destination
  from cte_flights
  where FlightNo between 25 and 100
 ),
 possible_dests AS (
  select Name AS Airline, FlightNo, Source, Destination
  from cte_flights
    join cte_airlines on Airline = Id
  where Destination in ('CVO', 'AED', 'ATO')
)
select b.*
from small_flightnos a
  join possible_dests b on (a.FlightNo = b.FlightNo)
```

SQL Fiddle

WITH RECURSIVE

`WITH RECURSIVE` allows a `WITH` query to refer to its *own* output. Example to sum the numbers from 1 to 100:

```
WITH RECURSIVE t(n) AS (
  SELECT 1
  UNION ALL
  SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

See also: [Solving the Traveling Salesman Problem with Postgres Recursive CTEs]

WITH RECURSIVE

`WITH RECURSIVE` allows a `WITH` query to refer to its *own* output. Example (from PostgreSQL documentation) to sum the numbers from 1 to 100:

```
WITH RECURSIVE t(n) AS (
   VALUES (1)
   UNION ALL
   SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

See also: [Solving the Traveling Salesman Problem with Postgres Recursive CTEs]

WITH RECURSIVE - Flights

CAL POLY

```
WITH RECURSIVE routes(the_source, the_dest, route, hops) as (
  SELECT Source as the_source, Destination,
  CAST(CONCAT(Source, '->', Destination) AS CHAR(150)) as route, 0 as hops
  FROM cte_flights
  WHERE Source = 'AMW'
  UNION
  SELECT routes.the_source, f.Destination,
  CONCAT(route, '->', f.Destination),
  routes.hops + 1 as hops
  FROM cte_flights f, routes
  WHERE f.Source = routes.the_dest and f.Destination <> 'AMW' AND hops < 3
)
SELECT * FROM routes
```

SQL Fiddle

SQL Views

[Virtual] View - Named SQL query whose results are not stored. Query will be run by the database as needed.

Materialized View - Query result stored as a physical table

Virtual Views

CAL POLY

- Virtual views are relations that are not physically stored like tables.
- Defined as a SQL `SELECT` statement
- Views can be queried as if they were regular tables.
- In certain cases, we can perform DML actions using virtual views

View Definition

Syntax to define a new view:

```
CREATE VIEW <view name> AS <view definition>;
```

Remove a view with:

```
DROP VIEW <view name>;
```

Dropping a view *does not* affect any underlying data.

Virtual View Example

CAL POLY

```
CREATE VIEW RoomSummary AS
SELECT Res.Code, Rm.RoomCode, LastName AS CustLastName,
  CheckIn, CheckOut,
  DATEDIFF(Checkout, CheckIn) AS Days,
  DATEDIFF(Checkout, CheckIn) * Rate AS TotalCharge,
  Adults + Kids AS GuestCount
FROM INN.reservations Res
  INNER JOIN INN.rooms Rm ON Res.Room = Rm.RoomCode
ORDER BY DATEDIFF(Checkout, CheckIn) * Rate DESC
```

Example View Usage

CAL POLY

```
-- Rooms with total charge > $2,500

SELECT RoomName, BasePrice, RS.*
FROM INN.rooms R
  INNER JOIN RoomSummary RS ON RS.RoomCode = R.RoomCode
WHERE TotalCharge > 2500
ORDER BY TotalCharge DESC
```

Updatable Views

A view may be updatable if defined using a "simple" SQL query. To allow this, a view must be defined using a limited set of SQL features. A few restrictions:

- `SELECT DISTINCT` not permitted
- `FROM` includes just one table
- `SELECT` list must include *all* required (`NOT NULL`) / key attributes

In other words, the view body must consist of and extremely simple query to permit DML statements "through" the view.

Materialized Views

If a view is used frequently or is very expensive to compute (typical for OLAP queries), it is often useful to *materialize* the view (not supported in MySQL) This causes the query results to be stored.

Downsides:

- Storage usage
- Materialized view must be *maintained* by the RDBMS: Each time a change is made to an underlying base table, the materialized data must be updated.

Materialized View Maintenance

CAL POLY

- Incremental
  - Changes are propagated to view *as they occur* in base tables
  - Possible for simple views, difficult as views become more complex
  - Downside: not always wise for frequently-updated base data (performance considerations)
- Periodic Refresh
  - Entire materialized view is reconstructed periodically (ie. each night)
  - Downside: data in materialized view is out of date

GRANT / REVOKE

Basic data access rules can be applied using SQL's "Data Control Language" statements:

- `GRANT` - Allow users/groups to perform certain actions on objects in the database
- `REVOKE` - Take away granted privileges

GRANT Syntax

CAL POLY

Basic syntax:

SELECT, INSERT, DELETE, UPDATE, etc.

```
GRANT <privilege(s)> ON <object(s)> TO <user(s)>;
```

Table or view name

Examples:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON reservations TO dclerk;
GRANT SELECT ON reservations TO taccountant;
```

(see documentation for additional options in MySQL)

Views with GRANT

One common use of views, along with the `GRANT` command is to apply basic access control rules within a database. Examples:

- Managers should see detail about employees in their own department only
- An HR analyst may change any salary except their own or supervisors'
- A payroll accountant should see only payroll-related accounts/transactions

Some databases extend this concept to support finer-grained security rules (eg. Oracle label security, Postgres row level security)

SQL Views / DCL - Summary

- A virtual views is nothing more than a named, stored SQL query
    - Remove from database with: `DROP VIEW <view name>;`
        - Dropping a view does not affect any data in physical tables
    - Can be used, along with data control statements (`GRANT` / `REVOKE`) to enforce data access rules in a multi-user database
    - To see tables vs. views in MySQL: `show full tables;`

- Materialized views (where supported) can offer significant performance benefits, offset by the problem of *maintenance*
    - A materialized view caches results of complex (slow) queries
    - Changes to base tables must be propagated to materialized views
    - Storage costs must be considered

SQL Views - Example

CAL POLY

BAKERY: List total sales amount for each food and flavor combination, along with the percent of sales for each flavor within its type of pastry (Twist, Meringue, Cake, Danish, etc.)

Flavor	Food	TotalSales	PctOfFoodSales
Almond	Twist	20.70	100
Chocolate	Meringue	25	60.8
Vanilla	Meringue	16.10	39.2
...	...	...	...

% should sum to 100 for each pastry type (a.k.a. food)