Using CTEs with Redshift

INTRODUCTION TO REDSHIFT

Jason Myers
Principal Architect

datacamp

Common table expressions (CTEs)

- Temporary result set
- Simplify queries
- Alternative to subqueries

Subquery and CTE structures

```
SELECT division_name,
       revenue_total
       -- Subquery for top 10 divions by revenue
  FROM (SELECT division_id,
               SUM(revenue) AS revenue_total
          FROM orders
         GROUP BY division_id
         ORDER BY revenue_total DESC
         LIMIT 10) top_ten_divisions_by_rev
 WHERE revenue_total > 100000;
```

```
-- CTE for top 10 divions by revenue
WITH top_ten_divisions_by_rev AS (
  SELECT division_id,
         SUM(revenue) AS revenue_total
    FROM orders
   GROUP BY division_id
   ORDER BY revenue_total DESC
   LIMIT 10)

-- Main query
SELECT division_id,
       revenue_total
       -- FROM our CTE
  FROM top_ten_divisions_by_rev
 WHERE revenue_total > 100000;
```

Multiple CTEs

```
-- Top 10 divisions by revenue CTE
WITH top_ten_divisions_by_rev AS(
  SELECT division_id,
         SUM(revenue) AS revenue_total
    FROM orders
   GROUP BY division_id
   ORDER BY revenue_total DESC
   LIMIT 10),
-- Division ID and Name CTE
division_names AS(
  SELECT id AS division_id,
         name AS division_name
    FROM divisions)
```

```
-- Main query
SELECT division_name,
       revenue_total
       -- FROM Top 10 CTE
  FROM top_ten_divisions_by_rev
  -- Joining so we can use the division name
  JOIN division_names USING (DIVISION_ID)
 WHERE revenue_total > 100000;
```

Multiple CTEs, CTE joins

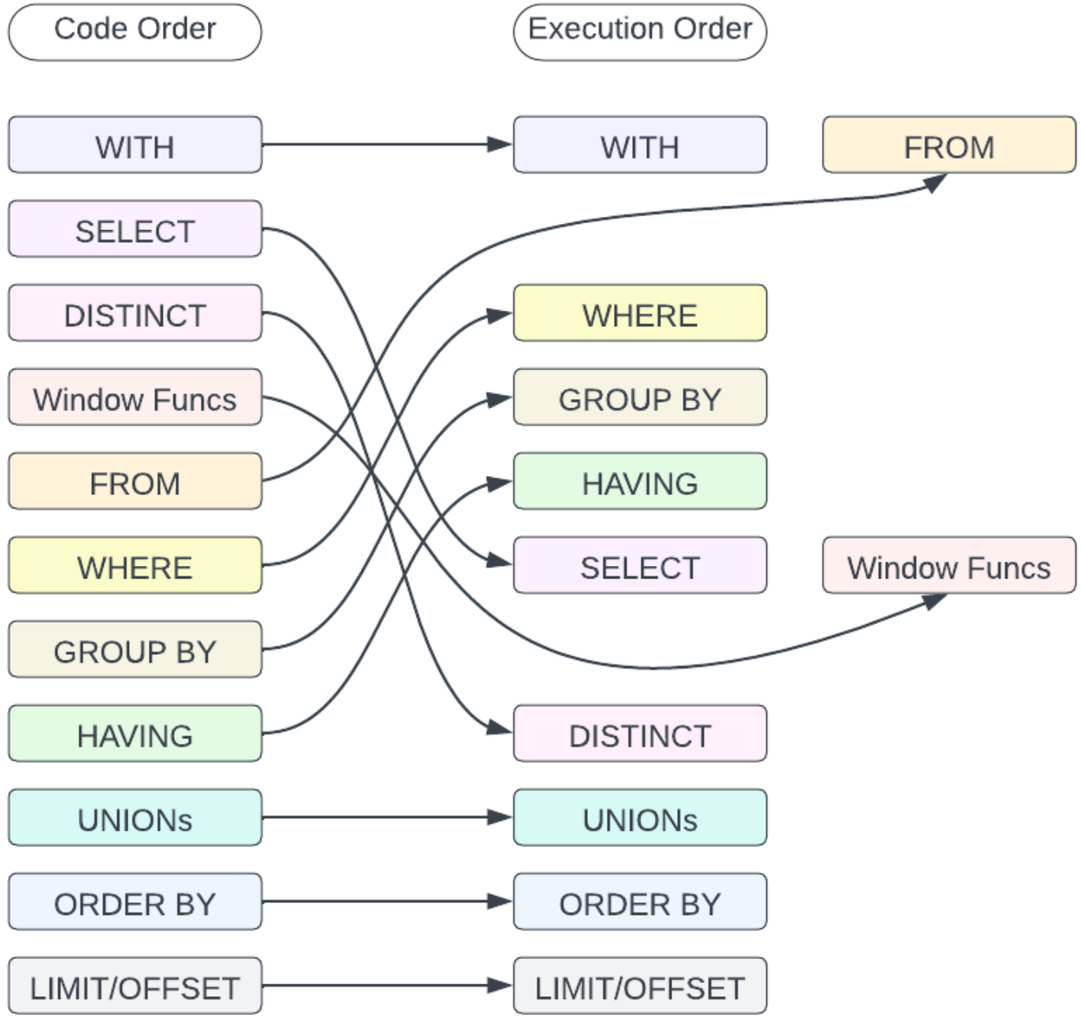
```
-- Selecting the division names
WITH division_names AS(
  SELECT id AS division_id,
         name AS division_name
    FROM divisions),
-- Selecting the top ten divisions by revenue
top_ten_divisions_by_rev AS(
  SELECT division_name,
         SUM(revenue) AS revenue_total
    FROM orders
    -- Joining in the division_names CTE
    JOIN divisions USING (division_id)
   GROUP BY division_name
   ORDER BY revenue_total DESC
   LIMIT 10);
```

Kernighan's Law

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?

CTE performance

- Same as subqueries
- Better than subqueries if reused



Let's practice!

INTRODUCTION TO REDSHIFT

Date and time functions

INTRODUCTION TO REDSHIFT

Jason Myers
Principal Architect

datacamp

Getting current date and time

- `SYSDATE` date and time at transaction start

```
-- Get current date and time
SELECT SYSDATE;
```

```
timestamp
============================
2024-01-27 20:05:55.976353
```

- `GETDATE()` date and time at statement start, requires parenthesis

```
-- Get current date and time
SELECT GETDATE();
```

```
timestamp
============================
2024-01-27 20:06:55.976353
```

Date and time function behavior

Watch out for leader node-only functions!

- `DATEDIFF` over `AGE`
- `GETDATE` / `SYSDATE` over leader-specific functions:
  - `CURRENT_TIME`
  - `CURRENT_TIMESTAMP`
  - `ISFINITE`
  - `LOCALTIME`
  - `LOCALTIMESTAMP`
  - `NOW`

Truncating dates and times

- `TRUNC` returns a date from a timestamp

```
-- Get current date based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT TRUNC(SYSDATE);
```

```
2024-01-27
```

- `DATE_TRUNC('datepart', timestamp)` truncates to a datepart like hour or day

```
-- Truncate to hour based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT DATE_TRUNC('minute', SYSDATE);
```

```
2024-01-27 20:05:55
```

[1] https://docs.aws.amazon.com/redshift/latest/dg/r_Dateparts_for_datetime_functions.html

Getting parts of dates and timestamps

- `DATE_PART(datepart, date or timestamp)`
  - extracts the requested part from a date or timestamp
- Can return more than `month`, `day`, `year`
  - Examples: `dayofweek`, `quarter`, `timezone`

```
-- Get current month based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT DATE_PART(month, SYSDATE);
```

```
1
```

```
-- Get current day of week based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT DATE_PART(dayofweek, SYSDATE);
```

```
6
```

Comparing dates and times

- `DATE_CMP(date_1, date_2)` relative comparison
  - Returns -1 if date_1 is earlier
  - Returns 0 if dates are equal
  - Returns 1 if date_1 is later
- Type specific functions
  - `DATE_CMP_TIMESTAMP`
  - `DATE_CMP_TIMESTAMPTZ`
  - `TIMESTAMP_CMP`
  - `TIMESTAMP_CMP_TIMESTAMPTZ`
  - `TIMESTAMPTZ_CMP`

```
-- Compare 5 dates from a table based on
-- SYSDATE of 2024-01-27 20:05:55.976353
  SELECT date_col,
         TRUNC(SYSDATE) AS current_date,
         DATE_CMP(date_col, SYSDATE)
    FROM combined_history_projections
ORDER BY date_col
   LIMIT 3;
```

```
 date_col   |  current_date | date_cmp
============|===============|==========
2024-01-26 | 2024-01-27    |       -1
2024-01-27 | 2024-01-27    |        0
2024-01-28 | 2024-01-27    |        1
```

Calculating differences

- `DATEDIFF(datepart, value_1, value_2)`
- Supports date, time, timetz, or timestamp in either position
  - Must contain the datepart
- Returns
  - a negative value if value_2 is earlier
  - Returns 0 if dates are equal
  - a positive value if value_2 is later

Using DATEDIFF

```
-- Days till end of first quarter based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT DATEDIFF(day,TRUNC(SYSDATE),'2024-03-31') AS days_diff;
```

```
days_diff
===========
64
```

Incrementing dates and times

- `DATEADD(datepart, quantity, value)`
- Supports date, time, timetz, or timestamp
- Quantity can be negative to subtract

```
-- Add week to a date based on
-- SYSDATE of 2024-01-27 20:05:55.976353
SELECT TRUNC(SYSDATE) AS todays_date,
       TRUNC(DATEADD(week, 1, SYSDATE)) AS next_weeks_date;
```

```
todays_date | next_weeks_date
============|================
2024-01-27  | 2024-02-03
```

Incrementing dates and times... gotchas

- Leap years by month return end of month

```
-- Add year by months to a date
SELECT DATEADD(month, 12, '2024-02-29');
```

```
2025-02-28 00:00:00
```

- Leap years by year return next day

```
-- Add year by year to a date
SELECT DATEADD(year, 1, '2024-02-29');
```

```
2025-03-01 00:00:00
```

Let's practice!

INTRODUCTION TO REDSHIFT

Windowing in Redshift

INTRODUCTION TO REDSHIFT

Jason Myers
Principal Architect

datacamp

Windows functions

- Operate on a window (partition) of data with a value for every row in that window
- Group functions aggregate result rows, but window functions do not
- Defined via an `OVER` clause

Three main concepts

- Partitioning - forming groups of rows (`PARTITION BY`)
- Ordering - order within each partition (`ORDER BY`)
- Framing - optional with additional restrictions on the rows.

Using windowing to calculate an average

```
SELECT division_id,
       sale_date,
       revenue,
       -- Calculate the average revenue
       AVG(revenue) OVER (
           -- By division for each year and month
           PARTITION BY division_id,
                        DATE_PART('year', sale_date)
                        DATE_PART('month', sale_date),
       ) AS month_avg_revenue
  FROM orders
 ORDER BY division_id,
          sale_date DESC;
```

Using windowing to calculate an average (cont)

division_id	sale_date	revenue	dept_month_avg_revenue
1	2024-01-23	350460	225500
1	2024-01-09	100540	225500
1	2023-12-15	231000	231000
1	2023-11-12	124000	68000
1	2023-11-07	75000	68000
1	2023-11-01	5000	68000
2	2024-01-10	500	500
2	2023-12-11	1000	16166.666666666667
2	2023-12-08	37000	16166.666666666667
2	2023-12-01	10500	16166.666666666667

Using lag for month over month windows

- LAG and LEAD help us get data for a row from one above(before) or below(after) it in the window according to the `ORDER BY` clause

```
SELECT division_id,
       DATE_PART('year', sale_date) AS sales_year,
       DATE_PART('month', sale_date) AS sales_month,
       -- Count records for the window
       COUNT(*) AS current_month_sales,
       -- Count the previous windows records
       LAG(COUNT(*), 1) OVER (
          -- For each division
          PARTITION BY division_id
              -- Ordered by year and month
              ORDER BY DATE_PART('year', sale_date),
                       DATE_PART('month', sale_date)
       ) AS prior_month_sales
```

Using lag for month over month windows (cont)

```
 FROM sales_data
-- Make sure to group by all the window clauses
GROUP BY division_id,
         sales_year,
         sales_month
ORDER BY division_id,
         sales_year DESC,
         sales_month DESC;
```

Using lag for month over month windows (results)

division_id	sales_year	sales_month	current_month_sales	prior_month_sales
1	2024	1	2	1
1	2023	12	1	3
1	2023	11	3	null
2	2024	1	1	3
2	2023	12	3	null

Ranking data within windows

- `RANK` allows us to rank a value over the window according to the `ORDER BY` clause starting with 1

```
SELECT division_id,
       sale_date,
       revenue,
       -- Calculate the rank for each sale in the window
       RANK() OVER (
           -- For each division
           PARTITION BY division_id
               -- Using revenue for the rank
               ORDER BY revenue desc
       ) as division_sales_rank
  FROM sales_data
 -- Put them in rank order by division
 ORDER BY division_id, division_sales_rank;
```

Ranking data within windows (results)

division_id	sale_date	revenue	division_sales_rank
1	2024-01-23	350460	1
1	2023-12-15	231000	2
1	2023-11-12	124000	3
1	2024-01-09	100540	4
1	2023-11-07	75000	5
1	2023-11-01	5000	6
2	2023-12-08	37000	1
2	2023-12-01	10500	2
2	2023-12-11	1000	3
2	2024-01-10	500	4

Let's practice!

INTRODUCTION TO REDSHIFT

Transactions

INTRODUCTION TO REDSHIFT

Jason Myers
Principal Engineer

Motivation for using transactions

```
SELECT name,
       priority,
  FROM data_log
       -- SYSDATE = 2024-02-07 00:17:24.259227
 WHERE intake_ts < SYSDATE;

SELECT name,
       data_size,
  FROM data_details
       -- SYSDATE = 2024-02-07 00:18:04.830527
 WHERE current_intake_date < SYSDATE;
```

Statement grouping example

`data_intake`

name	priority
idaho_monitoring_locations	1
idaho_samples	2
idaho_site_id	3

```
UPDATE data_intake
   SET priority=1
 WHERE name='idaho_samples';

UPDATE data_intake
   SET priority=2
 WHERE name='idaho_monitoring_locations';
```

Errored table results

`data_intake`

name	priority
idaho_monitoring_locations	1
idaho_samples	1
idaho_site_id	3

Transaction advantages and considerations

- Consistent data outcomes
- Requiring success or failure for a group of queries
- Concurrent operations

Default Execution Behavior

- Each SQL statement is a transaction!

Transactions Affect some Functions

- Set at start of transactions and stay the same
  - `SYSDATE`, `TIMESTAMP`, `CURRENT_DATE`

Some functions skirt around Transactions

- Set at each statement execution
  - `GETDATE`, `TIMEOFDAY`

Transactions Structure

- Opens with `BEGIN;` or `START TRANSACTION;`
- Contains one or more SQL statements with a semicolon after each one
- Closes with `END;` or `COMMIT;`
- NOTE: Semicolons matter

```
BEGIN;
  query1;
  query2;
END;
```

Getting consistent query results

```
-- Start a transaction
BEGIN;
SELECT name,
       priority,
  FROM data_log
       -- SYSDATE = 2024-02-07 00:17:24.259227
 WHERE intake_ts < SYSDATE;

SELECT name,
       data_size,
  FROM data_details
       -- SYSDATE = 2024-02-07 00:17:24.259227
 WHERE current_intake_date < SYSDATE;
-- End a transaction
END;
```

Function behavior in transactions

```
-- Start a transaction
BEGIN;
SELECT name,
       priority,
  FROM data_intake
       -- GETDATE = 2024-02-07 00:17:24.259227
 WHERE data_intake_ts < GETDATE();

 SELECT name,
       data_size,
  FROM data_details
       -- GETDATE = 2024-02-07 00:18:44.830527
 WHERE current_intake_date < GETDATE();
-- End a transaction
END;
```

Let's practice!

INTRODUCTION TO REDSHIFT