Week 4: Introduction to Database Management

Parch and Posey Database

Contents

Recap: In the Parch & Posey database there are five tables:

- web_events
- accounts
- orders
- sales_reps
- region

Figure 1 shows the ERD (entity relationship diagram) for Parch and Posey.

Reference: These notes are based on material from Mode Analytics on Udacity as well as w3schools.

Subqueries

Subqueries (also known as inner queries or nested queries) are a tool for performing operations in multiple steps. For example, if you wanted to take the sums of several columns, then average all of those values, you'd need to do each aggregation in a distinct step.

1. What is the lifetime average amount spent in terms of total_amt_usd for the top 10 total spending accounts?

In order to solve this query: (1) First we find the top 10 accounts in terms of highest `total_amt_usd`, and then (2) we find the average of these 10 amounts.

Let's start by finding the top 10 accounts:

```
SELECT a.name, sum(total_amt_usd) total_spent
FROM accounts a
JOIN orders o
  ON o.account_id = a.id
GROUP BY a.name
```

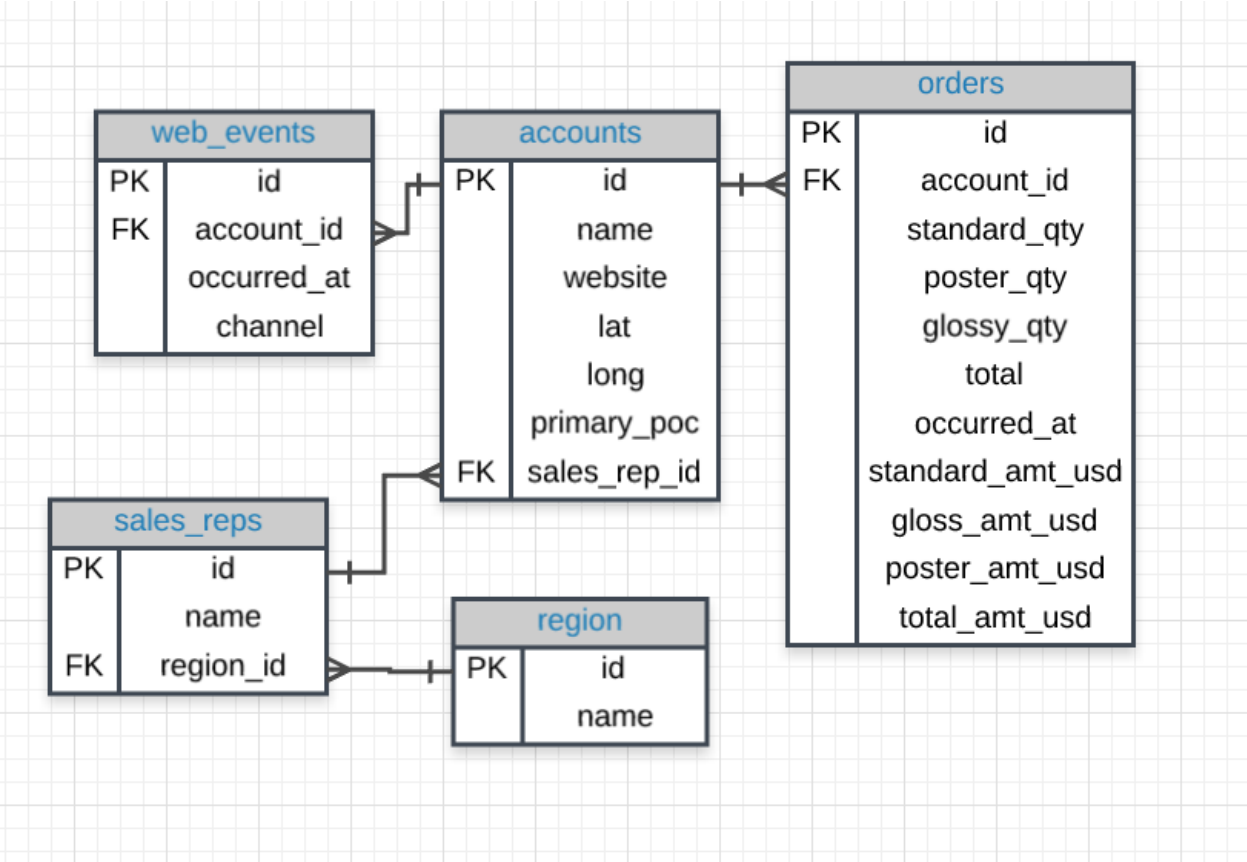

Figure 1: Parch and Posey ERD

```
ORDER BY total_spent DESC
LIMIT 10
```

name	total_spent
EOG Resources	382873.3
Mosaic	345618.6
IBM	326819.5
General Dynamics	300694.8
Republic Services	293861.1
Leucadia National	291047.2
Arrow Electronics	281018.4
Sysco	278575.6
Supervalu	275288.3
Archer Daniels Midland	272672.8

Then, find the average of their total amount spent:

```
SELECT avg(total_spent)
FROM
  (SELECT a.name, sum(total_amt_usd) total_spent
    FROM accounts a
    JOIN orders o
      ON o.account_id = a.id
    GROUP BY a.name
    ORDER BY total_spent DESC
    LIMIT 10
  ) tbl1
```

avg
304847

2. What is the lifetime average amount spent in terms of total_amt_usd for only the companies that spent (on average) more than the average of all orders.

In order to solve this query, we need to: (1) pull the average of all accounts in terms of total_amt_usd, and then (2) pull the accounts with more than this average amount, and finally (3) find the average of these values.

Let's start with the first query to pull the average of all accounts in terms of total_amt_usd

```
SELECT AVG(o.total_amt_usd) avg_all
FROM orders o
JOIN accounts a
ON a.id = o.account_id;
```

avg_all
3348.02

Now, let's pull the accounts with more than this average amount.

```
SELECT o.account_id, AVG(o.total_amt_usd)
FROM orders o
GROUP BY 1
HAVING AVG(o.total_amt_usd) > (SELECT AVG(o.total_amt_usd) avg_all
```

```
                                FROM orders o
                                JOIN accounts a
                                ON a.id = o.account_id)
                                LIMIT 5;
```

account_id	avg
4151	5236.645
4201	5365.058
3581	4857.855
1521	3612.052
1541	4828.861

Finally, let's find the average of these values:

```
SELECT AVG(avg_amt)
FROM (SELECT o.account_id, AVG(o.total_amt_usd) avg_amt
    FROM orders o
    GROUP BY 1
    HAVING AVG(o.total_amt_usd) > (SELECT AVG(o.total_amt_usd) avg_all
                                FROM orders o
                                JOIN accounts a
                                ON a.id = o.account_id)) temp_table;
```

avg
4721.14

3. For the customer that spent the most (in total over their lifetime as a customer) `total_amt_usd`, how many web_events did they have for each channel?

In order to solve this query, we need to: (1) first find the customer/account that spent the most, and then use it to find the number of web events asscoiated with that customer/account.

Let's start by finding that customer/account that spent the most:

```
SELECT name
FROM (
  SELECT
    a.name,
    sum(total_amt_usd) total_spent
  FROM accounts a
    JOIN orders o
      ON a.id = o.account_id
  GROUP BY a.name
  ORDER BY total_spent DESC
  LIMIT 1
) tbl1
```

name
EOG Resources

Then, use that customer to find the web_events for them:

```
SELECT a.name, w.channel, count(*)
FROM accounts a
```

```
JOIN web_events w
  ON w.account_id = a.id
GROUP BY a.name, w.channel
HAVING a.name = (
  SELECT name
FROM (
  SELECT
    a.name,
    sum(total_amt_usd) total_spent
  FROM accounts a
    JOIN orders o
      ON a.id = o.account_id
  GROUP BY a.name
  ORDER BY total_spent DESC
  LIMIT 1
) tbl1

)
```

name	channel	count
EOG Resources	adwords	12
EOG Resources	banner	4
EOG Resources	direct	44
EOG Resources	facebook	11
EOG Resources	organic	13
EOG Resources	twitter	5

4. For the region with the largest sales total_amt_usd, how many total orders were placed?

In order to solve this query, we need to: (1) find the region with the highest total `total_amt_usd`, and then (2) find the total orders (count) for that region.

Let's start with the first query to find the region with the highest total `total_amt_usd`:

```
SELECT name
FROM (
       SELECT r.name, sum(o.total_amt_usd) total_spent
        FROM region r
          JOIN sales_reps s
            ON r.id = s.region_id
          JOIN accounts a
            ON a.sales_rep_id = s.id
          JOIN orders o
            ON o.account_id = a.id
        GROUP BY r.name
        ORDER BY total_spent DESC
        LIMIT 1
      ) tbl1
```

name
Northeast

Then, find the total orders (count) for that region:

```
SELECT r.name, count(*)
FROM region r
          JOIN sales_reps s
            ON r.id = s.region_id
          JOIN accounts a
            ON a.sales_rep_id = s.id
          JOIN orders o
            ON o.account_id = a.id
GROUP BY r.name
HAVING r.name = (SELECT name
                 FROM (
                         SELECT
                           r.name,
                           sum(o.total_amt_usd) total_spent
                         FROM region r
                           JOIN sales_reps s
                             ON r.id = s.region_id
                           JOIN accounts a
                             ON a.sales_rep_id = s.id
                           JOIN orders o
                             ON o.account_id = a.id
                         GROUP BY r.name
                         ORDER BY total_spent DESC
                         LIMIT 1
                         ) tbl1
                 )
```

name	count
Northeast	2357

5. For the account that purchased the most (in total over their lifetime as a customer) standard_qty paper, how many accounts still had more in total purchases?

In order to solve this query: (1) First, we find the account that had the most standard_qty paper. Then, (2) use this to pull all the accounts with more total sales, and finally (3) get the count with just another simple subquery.

Let's start with the first query to find the account that had the most standard_qty paper:

```
SELECT a.name account_name, SUM(o.standard_qty) total_std
FROM accounts a
JOIN orders o
ON o.account_id = a.id
GROUP BY a.name
ORDER BY SUM(o.standard_qty) DESC
LIMIT 1;
```

account_name	total_std
Core-Mark Holding	41617

Next, use the result above to pull all the accounts with more total sales

```
SELECT a.name
FROM orders o
```

```
JOIN accounts a
ON a.id = o.account_id
GROUP BY a.name
HAVING SUM(o.total) > (SELECT tot_std
                  FROM (SELECT a.name act_name, SUM(o.standard_qty) tot_std
                        FROM accounts a
                        JOIN orders o
                        ON o.account_id = a.id
                        GROUP BY a.name
                        ORDER BY SUM(o.standard_qty) DESC
                        LIMIT 1) tbl1);
```

name
Leucadia National
EOG Resources
Mosaic
Core-Mark Holding
IBM
General Dynamics

Finally, get the count with just another simple subquery:

```
SELECT COUNT(*)
FROM (SELECT a.name
      FROM orders o
      JOIN accounts a
      ON a.id = o.account_id
      GROUP BY a.name
      HAVING SUM(o.total) > (SELECT tot_std
                  FROM (SELECT a.name act_name, SUM(o.standard_qty) tot_std, SUM(o.total) total
                        FROM accounts a
                        JOIN orders o
                        ON o.account_id = a.id
                        GROUP BY a.name
                        ORDER BY SUM(o.standard_qty) DESC
                        LIMIT 1) tbl1)
            ) tbl2;
```

count
6

6. Provide the name of the sales_rep in each region with the largest amount of total_amt_usd sales.

In order to solve this query, we can think of it as two subqueries: (1) One to find the `total_amt_usd` totals associated with each sales rep in each region in which they were located, and (2) a query to find the max for each region. Once you have the results of both, then we can join them to get the name of the sales_rep in each region with the largest amount of total_amt_usd sales.

Let's start with the first query to find the `total_amt_usd` totals associated with each sales rep in each region in which they were located:

```
SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
```

```
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY 1,2
ORDER BY 3 DESC
LIMIT 5
```

rep_name	region_name	total_amt
Earlie Schleusner	Southeast	1098137.7
Tia Amato	Northeast	1010690.6
Vernita Plump	Southeast	934212.9
Georgianna Chisholm	West	886244.1
Arica Stoltzfus	West	810353.3

Now, let's run the second query to find the max for each region:

```
SELECT region_name, MAX(total_amt) total_amt
     FROM(SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt
             FROM sales_reps s
             JOIN accounts a
             ON a.sales_rep_id = s.id
             JOIN orders o
             ON o.account_id = a.id
             JOIN region r
             ON r.id = s.region_id
             GROUP BY 1, 2) t1
     GROUP BY 1
     LIMIT 10;
```

region_name	total_amt
Midwest	675637.2
Southeast	1098137.7
West	886244.1
Northeast	1010690.6

Now, we can join the two tables based on which the amount for the sales_person matches the maximum amount for each region:

```
SELECT  t3.region_name, t3.rep_name, t3.total_amt
FROM(SELECT region_name, MAX(total_amt) total_amt
     FROM(SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt
             FROM sales_reps s
             JOIN accounts a
             ON a.sales_rep_id = s.id
             JOIN orders o
             ON o.account_id = a.id
             JOIN region r
             ON r.id = s.region_id
             GROUP BY 1, 2) t1
     GROUP BY 1) t2
```

```
JOIN (SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt
     FROM sales_reps s
     JOIN accounts a
     ON a.sales_rep_id = s.id
     JOIN orders o
     ON o.account_id = a.id
     JOIN region r
     ON r.id = s.region_id
     GROUP BY 1,2
     ORDER BY 3 DESC) t3
ON t3.region_name = t2.region_name AND t3.total_amt = t2.total_amt;
```

region_name	rep_name	total_amt
Southeast	Earlie Schleusner	1098137.7
Northeast	Tia Amato	1010690.6
West	Georgianna Chisholm	886244.1
Midwest	Charles Bidwell	675637.2

In fact, we didn't have to go with the subquery route for the previous question. All we have to do is to find out the total sales for each combination of salespeople and region, sort the result in descending order based on the total amount spent, and the slice the result to 4 rows since we know that there are only 4 regions (I wanted to execute the subqueries before for learning purposes:

```
SELECT r.name region_name, s.name rep_name, sum(o.total_amt_usd) total_spent
FROM orders o
  JOIN accounts a
    ON o.account_id = a.id
  JOIN sales_reps s
    ON s.id = a.sales_rep_id
  JOIN region r
    ON r.id = s.region_id
GROUP BY r.name, s.name
ORDER BY total_spent DESC
LIMIT 4
```

region_name	rep_name	total_spent
Southeast	Earlie Schleusner	1098137.7
Northeast	Tia Amato	1010690.6
Southeast	Vernita Plump	934212.9
West	Georgianna Chisholm	886244.1

Subqueries using 'WITH'

We can also use *Common Table Expressions* (also know as CTEs) to provide intermediate, or auxiliary statements for a larger, more complex query. A CTE is a temporary result set created in memory on the database that can be referenced in another SQL statement.They do not exist once the main query finishes executing.

Let's look at some examples of CTEs:

7. What is the lifetime average amount spent in terms of `total_amt_usd` for the top 10 total spending accounts?

```sql
WITH t1 AS (
  SELECT a.name, sum(total_amt_usd) total_spent
  FROM accounts a
  JOIN orders o
    ON a.id = o.account_id
  GROUP BY a.name
  ORDER BY total_spent DESC
  LIMIT 10
)
SELECT avg(total_spent)
FROM t1
```

avg
304847

We can also use multiple CTEs in the same query. For instance, in the below query, we are using both `tbl1` and `tbl2` as CTEs in the query.

8. What is the lifetime average amount spent in terms of `total_amt_usd` for only the companies that spent more than the average of all orders.

```sql
WITH tbl1 as(
  SELECT avg(total_amt_usd) avg_spent
  FROM orders
),
tbl2 as(
  SELECT a.name, avg(total_amt_usd) total_spent
  FROM accounts a
  JOIN orders o
    ON o.account_id = a.id
  GROUP BY a.name
  HAVING avg(total_amt_usd) > (SELECT * FROM tbl1)
)
SELECT avg(total_spent)
FROM tbl2
```

avg
4721.14

9. For the customer that spent the most (in total over their lifetime as a customer) `total_amt_usd`, how many web_events did they have for each channel?

```sql
with tbl1 AS (
    SELECT a.name, sum(total_amt_usd) total_spent
    FROM orders o
    JOIN accounts a
      ON o.account_id = a.id
    GROUP BY a.name
    ORDER BY total_spent DESC
    LIMIT 1)
SELECT a.name, w.channel, count(*) num_events
FROM web_events w
JOIN accounts a
  ON a.id = w.account_id and a.name = (SELECT name FROM tbl1)
```

```
GROUP BY a.name, w.channel
ORDER BY num_events DESC
```

name	channel	num_events
EOG Resources	direct	44
EOG Resources	organic	13
EOG Resources	adwords	12
EOG Resources	facebook	11
EOG Resources	twitter	5
EOG Resources	banner	4

10. For the region with the largest sales total_amt_usd, how many total orders were placed?

```
with tbl1 as (
    SELECT r.name, sum(total_amt_usd) total_spent
    FROM region r
    JOIN sales_reps s
      ON s.region_id = r.id
    JOIN accounts a
      ON s.id = a.sales_rep_id
    JOIN orders o
      ON o.account_id  = a.id
    GROUP BY r.name
    ORDER BY total_spent DESC
    LIMIT 1)
SELECT r.name, count(*) num_orders
FROM region r
JOIN sales_reps s
    ON s.region_id = r.id
JOIN accounts a
    ON s.id = a.sales_rep_id
JOIN orders o
    ON o.account_id  = a.id and r.name = (SELECT name FROM tbl1)
GROUP BY r.name
```

name	num_orders
Northeast	2357

11. For the account that purchased the most (in total over their lifetime as a customer) standard_qty paper, how many accounts still had more in total purchases?

```
with tbl1 as (
    SELECT a.name, sum(standard_qty) total_std_qty
    FROM accounts a
    JOIN orders o
      ON a.id = o.account_id
    GROUP BY a.name
    ORDER BY total_std_qty DESC
    LIMIT 1)
SELECT a.name, sum(total) total_paper_qty
FROM accounts a
JOIN orders o
  ON a.id = o.account_id
GROUP BY a.name
```

```
HAVING sum(total) > (SELECT total_std_qty FROM tbl1)
```

name	total_paper_qty
Leucadia National	42358
EOG Resources	56410
Mosaic	49246
Core-Mark Holding	44750
IBM	47506
General Dynamics	43730

12. Provide the name of the sales_rep in each region with the largest amount of total_amt_usd sales.

```
with tbl1 as(
    SELECT s.name sales_person, r.name region_name, sum(total_amt_usd) total_spent
    FROM region r
    JOIN sales_reps s
      ON s.region_id = r.id
    JOIN accounts a
      ON a.sales_rep_id = s.id
    JOIN orders o
      ON o.account_id = a.id
    GROUP BY s.name, r.name
    ORDER BY r.name, total_spent DESC
  ),
tbl2 as(
    SELECT region_name, max(total_spent) max_total_spent
    FROM tbl1
    GROUP BY region_name
)
SELECT tbl1.region_name, max_total_spent
FROM tbl1
JOIN tbl2
  ON tbl1.region_name = tbl2.region_name and tbl1.total_spent = tbl2.max_total_spent
ORDER BY max_total_spent DESC
LIMIT 5;
```

region_name	max_total_spent
Southeast	1098137.7
Northeast	1010690.6
West	886244.1
Midwest	675637.2

Database Management Commands

We will now be discussing briefly some common database management operations you should be familiar with in your work as a data analyst, engineer, or scientist. Often, you may find that you have to actually load the data you want to work with into a database and manage the tables directly.

Note: make sure for the following sections you are logged in with you personal user roles. For example, my personal user role is `ychen220`. The role that you have been using so far - 'dso-5

The first command to understand is the `CREATE TABLE` command. This will create a table in the current schema. Note that your current schema is usually set to the `public` schema by default.

Let's create a table called `web_events` in your new schema. Ensure you are logged in as your personal user and run the following command:

```
CREATE TABLE ychen220.web_events (
    id INTEGER,
    account_id INTEGER,
    occurred_at TIMESTAMP,
    channel TEXT
);
```

This creates a table called `web_events` inside your own personal schema called `username`.

Notice now in pgAdmin 4 under the `ychen220` schema, a new table called `web_events` is created. We specify in parenthesis the column name (for example `id`), and then the data type of the column. Each column, or field, must have one, and only one, data type. Common data types include:

- `INTEGER`
- `DECIMAL`
- `DOUBLE` (a double precision number value)
- `DATE` (a date on the calendar)
- `TIMESTAMP` (an exact moment in time)
- `TEXT`, `VARCHAR`, `CHAR` (text data types)
- `NUMERIC`
- `JSON` (we'll learn about this in the following weeks)

Tables

Populating the Table

The table `web_events` is completely empty. We'll insert a record into it by using the `INSERT` command:

```
INSERT INTO web_events VALUES (1,1001,'2015-10-06 17:13:58','direct');
```

We specify the table that we want to insert into (`web_events`) followed by the keyword `VALUES`. Then, in parenthesis, we specify a tuple of values to insert. Notice that the order of the columns and the order of the values must match. For instance, I am inserting for the field `id` a value of `1`, and for `account_id` a value of `1001`.

If we now run a `SELECT` query to get the contents of this table, you'll see that we get one row:

```
SELECT *
FROM ychen220.web_events;
```

id	account_id	occurred_at	channel
1	1001	2015-10-06 17:13:58	direct

We must match the data types of the insert tuple with the data types defined in the table's DDL (data definition language) statement. For instance, we can't try to insert the following into `ychen220.web_events`:

```
INSERT INTO web_events VALUES ('Yu Chen',1001,'2015-10-06 17:13:58','direct');
```

We get the following error:

```
ERROR:  invalid input syntax for type integer: "Yu Chen"
LINE 1: INSERT INTO web_events VALUES ('Yu Chen',1001,'2015-10-06 17...
```

This is because the `id` column expects an `INTEGER` type, but `'Yu Chen'` is a `TEXT` type.

We can also insert more than one record into a database table. Let's create a new table called

```
CREATE TABLE ychen220.web_events_bulk_insert (
    id INTEGER,
    account_id INTEGER,
    occurred_at TIMESTAMP,
    channel TEXT
);

-- insert 4 records at one time into the database table (bulk insert)
INSERT INTO ychen220.web_events_bulk_insert
VALUES (1, 1001, '2015-10-06 17:13:58', 'direct'),
       (2, 1001, '2015-11-05 03:08:26', 'direct'),
       (3, 1001, '2015-12-04 03:57:24', 'direct'),
       (4, 1001, '2016-01-02 00:55:03', 'direct');
```

Here, we are inserting 4 records at the same time into the `web_events_bulk_insert` table. If you have a choice between inserting one record, versus inserting many records, try to use bulk inserts. Bulk inserts are significantly faster than individually attempting to insert one record at a time.

```
SELECT *
FROM ychen220.web_events_bulk_insert;
```

id	account_id	occurred_at	channel
1	1001	2015-10-06 17:13:58	direct
2	1001	2015-11-05 03:08:26	direct
3	1001	2015-12-04 03:57:24	direct
4	1001	2016-01-02 00:55:03	direct

We can also create tables based on the result set of other SQL queries. Here, we will create a table called `facebook_channel_web_events` that contains only web events sourced from `facebook`.

```
CREATE TABLE ychen220.facebook_channel_web_events AS
SELECT *
FROM public.web_events WHERE channel = 'facebook';
```

Now, if we can run a query to confirm that the only value for `channel` inside this table is `'facebook'`:

```
SELECT DISTINCT channel
FROM ychen220.facebook_channel_web_events
```

channel
facebook

To delete a row from a table, we can use the `DELETE` command. For example, if we wish to delete all the rows from `facebook_channel_web_events`, we can run

```
DELETE FROM ychen220.facebook_channel_web_events;
```

Now, if you query from this table, you'll notice that the entire table is empty. **Note: Be extremely careful with `DELETE`. In general, most database administrators will give permissions for the `DELETE` command to only a few certain individuals, since running a `DELETE FROM ...` arbitrarily could mean that all of the company's orders are wiped.

This time, let's create another table and then delete only the rows where `channel` is `facebook`:

```sql
CREATE TABLE ychen220.web_events_without_facebook AS
SELECT *
FROM public.web_events;

-- delete all rows where web events is equal to Facebook
DELETE FROM ychen220.web_events_without_facebook
WHERE channel = 'facebook';
```

Updating Rows in a Table

We can also update individual rows in a table using the `UPDATE` command. For instance, let's create a new table called `new_channel_names` that is a copy of the `public.web_events` table.

```sql
CREATE TABLE ychen220.new_channel_names AS
SELECT *
FROM public.web_events;

-- delete all rows where web events is equal to Facebook
UPDATE ychen220.new_channel_names
SET channel = 'FB'
WHERE channel = 'facebook'
```

This will update all rows where the `channel` value is `facebook` and set that channel value to `FB`.

Now when we run a query on `new_channel_names`, notice that we see `FB` is one of the channels:

```sql
SELECT channel, COUNT(*)
FROM ychen220.new_channel_names
GROUP BY 1
```

channel	count
twitter	474
adwords	906
FB	967
organic	952
banner	476
direct	5298

Deleting Tables

Finally, we can delete tables entirely by using the `DROP TABLE` command. For example, this command will drop the table that we just created:

```sql
DROP TABLE ychen220.new_channel_names;
```

Make sure you understand the difference between `DROP TABLE` and `DELETE FROM`. The `DROP TABLE` command will delete the entire table from the database. The `DELETE FROM` will delete certain (and if no `WHERE` clause is specified, all) rows from the table.

Views

Often, we have to write very complex queries to answer commonly asked business questions. For instance, we might be asked to populate a dashboard that should be updated in real time the total orders by the top 10 accounts for Parch & Posey. Let's see how we can create this real-time dashboard.

First, we'll copy over the data from the `accounts` and `orders` table into our own schema:

```sql
CREATE TABLE ychen220.accounts AS
SELECT *
FROM public.accounts;

CREATE TABLE ychen220.orders AS
SELECT *
FROM public.orders;
```

Notice that I am explicitly providing the schema names (`public` and `ychen220`) since I am now working with multiple tables with the same names.

Now that we have our base tables set up, first, we'll write the query to get the top 10 accounts based on order counts:

```sql
SELECT a.name                AS account_name,
       COUNT(DISTINCT o.id) AS order_count
FROM ychen220.accounts a
         JOIN ychen220.orders o ON a.id = o.account_id
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

The gets us the following result set, which does indeed get the top 10 accounts based on order count:

account_name	order_count
Leucadia National	71
Sysco	68
Supervalu	68
Arrow Electronics	67
Mosaic	66
Archer Daniels Midland	66
General Dynamics	66
Fluor	65
Philip Morris International	65
United States Steel	65

We see that the number one account is `Leucadia National`, followed by `Sysco`. Since we know that we are going to using the query frequently, and other analysts will reference this same query in other projects, let's save this as a `VIEW`:

```sql
CREATE VIEW ychen220.top_ten_accounts AS
SELECT a.name                AS account_name,
       COUNT(DISTINCT o.id) AS order_count
FROM ychen220.accounts a
         JOIN ychen220.orders o ON a.id = o.account_id
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

When we run this command, we will see that a new type of object is created inside the `ychen220` schema: a view object. A view object looks and behaves very similarly to a table. For instance, we can query from a view just like we can query from tables:

```sql
SELECT *
FROM ychen220.top_ten_accounts;
```

account_name	order_count
Leucadia National	71
Sysco	68
Supervalu	68
Arrow Electronics	67
Mosaic	66
Archer Daniels Midland	66
General Dynamics	66
Fluor	65
Philip Morris International	65
United States Steel	65

Notice that we get the same results as we did when we ran our `GROUP BY` initial SQL query! We can even create another view from the results of the `ychen220.top_ten_accounts` view. For instance, let's create a view that always returns the number average number of orders for our top 10 accounts:

```
CREATE VIEW ychen220.average_orders_for_top_accounts AS
SELECT AVG(order_count)
FROM ychen220.top_ten_accounts;
```

Then, when we query from the view `average_orders_for_top_accounts`, we get that average:

```
SELECT *
FROM ychen220.average_orders_for_top_accounts;
```

avg
66.7

The difference, however, between a **view** and a **table**, however, is that a view is not persisted to disk, or saved by the database permanently. A table's data is actually saved on the database server's disk space.

A view is simply a shortcut to query defined by the view. When we run

```
SELECT *
FROM ychen220.average_orders_for_top_accounts;
```

We are actually running

```
SELECT *
FROM  (
  SELECT AVG(order_count)
  FROM ychen220.top_ten_accounts) average_orders_for_top_accounts
```

And since `top_ten_accounts` is actually a view itself, then the database is actually running

```
SELECT *
FROM ( -- start of the average_orders_for_top_accounts view
         SELECT AVG(order_count)
         FROM ( -- start of the top_ten_accounts view
                  SELECT a.name               AS account_name,
                         COUNT(DISTINCT o.id) AS order_count
                  FROM ychen220.accounts a
                           JOIN ychen220.orders o ON a.id = o.account_id
                  GROUP BY 1
                  ORDER BY 2 DESC
```

```
                    LIMIT 10
                ) top_ten_accounts) average_orders_for_top_accounts
```

So make sure you understand the chain of information here. We have the `accounts` and `orders` tables that feed into the `top_ten_accounts` view, which is then used to generate the `average_orders_for_top_accounts` view itself.

So what is the point of a view, if they are simply shortcuts for queries themselves? One of the primary benefits is that they allow for **dynamically updated result sets**. For instance, let's go into the `orders` table and add in an extra order for the account `Sysco`:

```
INSERT INTO ychen220.orders VALUES
(-1,1561,'2016-08-14 07:39:19',272,48,5,325,1357.28,359.52,40.6,1757.4);
```

Note that `1561` is the account ID for `Sysco`, and `-1` is the order ID. I'm setting it as `-1` so it is easy to identify and delete later.

Now that we've inserted this extra record, let's now query our `top_ten_accounts` view:

```
SELECT *
FROM ychen220.top_ten_accounts;
```

Notice that now `Sysco`'s order account has now incremented to `69`, and now it is now in sole possession of 2nd place, instead of being tied with `Supervalu`.

account_name	order_count
Leucadia National	71
Sysco	69
Supervalu	68
Arrow Electronics	67
Mosaic	66
Archer Daniels Midland	66
General Dynamics	66
Fluor	65
Philip Morris International	65
United States Steel	65

We did not have to rerun any queries or update anything - our `top_ten_accounts` view automatically adjusted to incorporate the new information. This is the beauty of views - they are always updated in real time, so the information they present is not stale. If we had instead made our `top_ten_accounts` view a table, we'd have to update the table for the new top ten accounts to appear.

Dropping Views

We can drop views just like drop tables:

```
DROP VIEW IF EXISTS ychen220.top_ten_accounts;
```

The above command will check if the `ychen220.top_ten_accounts` view exists, and drop it if it does exist. If you do not specify `IF EXISTS`, then the command will throw an error if the `top_ten_accounts` view does not exist.

However, notice what happens if you attempt to actually drop `top_ten_accounts`:

```
ERROR:  cannot drop view ychen220.top_ten_accounts because other objects depend on it
DETAIL:  view ychen220.average_orders_for_top_accounts depends on view ychen220.top_ten_accounts
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

We have established a dependency between the different objects in our `ychen220` schema: the `average_orders_for_top_accounts` view depends on the `top_ten_accounts` view existing. Therefore, we get an error because PostgreSQL will not allow us to drop a view that is being used by another object.

Therefore, therefore we can use

```
DROP VIEW ychen220.top_ten_accounts CASCADE;
```

This tells PostgreSQL to drop `top_ten_accounts` but also drop any objects that depend on this view, such as `average_orders_for_top_accounts`.

Materialized Views

The benefit of views is that they are not saved to disk, and therefore are dynamic. The drawback of views is that they are not saved to disk, and therefore can be expensive and lengthy to run for very large queries and computations.

For instance, imagine if you worked at Uber, and required the following computation to show the number of rides per state:

```
SELECT c.residential_state, COUNT(DISTINCT ride.id) AS rides, COUNT(DISTINCT )
FROM customer c
JOIN ride r ON c.id = r.customer_id
GROUP BY 1
ORDER BY 2 DESC
```

There are likely to be tens, perhaps hundreds of millions of rides you have to scan through. You want to "cache" the result of this query so that the database does not constantly having to perform this expensive computation.

You can save the query as a **materialized view**:

```
CREATE MATERIALIZED VIEW rides_per_state AS
SELECT c.residential_state, COUNT(DISTINCT ride.id) AS rides, COUNT(DISTINCT )
FROM customer c
JOIN ride r ON c.id = r.customer_id
GROUP BY 1
ORDER BY 2 DESC
```

A materialized view is view's result set saved to disk. Therefore it operates extremely similarly to a table. However, when data in the `customer` or `ride` tables are updated (and new rides or customers are entered into the system, the data in `rides_per_state` is not updated dynamically). You'll get the same results as when you first created the materialized view. To update the materialized view with the new data, you'll have to run

```
REFRESH MATERIALIZED VIEW rides_per_state;
```

Let's take a look at an example from Parch & Posey. Let's pretend that there's millions upon millions of web events, such that it's difficult to compute the top 3 channels in a reasonable amount of time and it takes a large amount of computing power to do so. We'll create a materialized view to cache the results:

```
CREATE MATERIALIZED_VIEW ychen220.top_channels AS
SELECT channel, COUNT(*) AS num_events
FROM public.web_events
GROUP BY 1
ORDER BY 2 DESC
LIMIT 3
```

We can then query this materialized view just like any other view or table:

```
SELECT *
FROM ychen220.top_channels;
```

channel	num_events
direct	5298
facebook	967
organic	952

However, when new data is added to `web_events`, we'll need to run the `REFRESH` command to make sure that the data in the materialized view is kept up to date:

```
REFRESH MATERIALIZED VIEW ychen220.top_channels;
```