ADVANCED AGGREGATES

INTRODUCTION

At the heart of every great business decision is data. Since most businesses store critical data in SQL databases, a deep understanding of SQL is a necessary skill for every data analyst.

Chief among data analysis tasks is *data aggregation*, the grouping of data to express in summary form. We'll be working with **SpeedySpoon**, a meal delivery app. The folks at SpeedySpoon have asked us to look into their deliveries and help them optimize their process.

This course was developed in partnership with our good friends at Periscope Data. If you're new to SQL, we recommend you do this course first.

1. Complete each query by replacing the comments `/**/` with SQL code. We'll start by looking at SpeedySpoon's data. The `orders` table has a row for each order of a SpeedySpoon delivery. It says when the order took place, and who ordered it.

   Add code to the `select` statement to select all columns in the `orders` table.

```
select *
from /**/
order by id
limit 100;
```

Note that the `order` and `limit` clauses keep the data organized.

```
select *
from orders
order by id
limit 10;
```

2. The `order_items` table lists the individual foods and their prices in each `order`. Complete the query to select all columns in the `order_items` table.

```
select *
from /**/
order by id
limit 100;
```

```
select *
from order_items
order by id
limit 10;
Daily Counts
```

DAILY COUNTS

Nice work! Now that we have a good handle on our data, let's dive into some common business queries. We'll begin with the Daily Count of orders placed. To make our Daily Count metric, we'll focus on the `date` function and the `ordered_at` field.

To get the Daily Metrics we need the *date*. Most dates in databases are *timestamps*, which have hours and minutes, as well as the year, month, and day. Timestamps look like `2015-01-05 14:43:31`, while dates are the first part: `2015-01-05`.

We can easily select the date an item was ordered at with the `date` function and the `ordered_at` field:

```
select date(ordered_at)
from orders;
```

1. Let's get a Daily Count of orders from the `orders` table. Complete the query using the `date` function to cast the timestamps in `ordered_at` to dates.

```
select /**/
from orders
order by 1
limit 100;
```

The `order by 1` statement is a shortcut for `order by date(ordered_at)`. The `1` refers to the first column.

```
select date(ordered_at) as date
from orders
order by 1
limit 10;
```

DAILY COUNT 2

Great! Now that we can convert timestamps to dates, we can count the Orders Per Date. Generally, when we count all records in a table we run a query with the `count` function, as follows:

```
select count(1)
from users;
```

This will treat all rows as a single group, and return one row in the result set - the total count.

To count orders by their dates, we'll use the `date` and `count` functions and pair them with the `group by` clause. Together this will count the records in the table, grouped by date.
For example, to see the user records counted by the date created, we use the `date` and `count` functions and `group by` clause as follows:

```
select date(created_at), count(1)
from users
group by date(created_at)
```

1. Use the `date` and `count` functions and `group by` clause to count and group the orders by the dates they were `ordered_at`.

```
select /**/
from orders
group by 1
order by 1;
```

We now have the daily order count!

The `order by 1` and `group by 1` statements are shortcuts for `order by date(ordered_at)` and `group by date(ordered_at)`.

```
select date(ordered_at) as date, count(distinct date(ordered_at)) as orders
from orders
group by 1
limit 10;
```

DAILY REVENUE

We have the Daily Count of orders, but what we really want to know is revenue. How much money has **SpeedySpoon** made from orders each day?

1. We can make a few changes to our Daily Count query to get the revenue. First, instead of using `count(1)` to count the rows per date, we'll use `round(sum(amount_paid), 2)` to add up the revenue per date. Complete the query by adding revenue per date.

   Second, we need to join in the `order_items` table because that table has an `amount_paid` column representing revenue. Complete the query by adding a `join` clause where `orders.id = order_items.order_id`.

```
select date(ordered_at), /**/
from orders
    /**/ order_items on
      orders.id = order_items.order_id
group by 1
order by 1;
```

Note that the `round` function rounds decimals to digits, based on the number passed in. Here `round(..., 2)` rounds the sum paid to two digits.

```
select date(ordered_at) as date, round(sum(amount_paid), 2) as revenue
from orders
    join order_items
    on orders.id = order_items.order_id
```

```
group by 1
order by 1
limit 10;
```

2. Nice. Now with a small change, we can find out how much we're making per day for any single dish. What's the daily revenue from customers ordering kale smoothies? Complete the query by using a where clause to filter the daily sums down to orders where the `name = 'kale-smoothie'`.

```
select date(ordered_at), round(sum(amount_paid), 2)
from orders
    join order_items on
      orders.id = order_items.order_id
where /**/
group by 1
order by 1;
```

```
select date(ordered_at) as date, round(sum(amount_paid), 2) as revenue
from orders
    join order_items on
      orders.id = order_items.order_id
where name = 'kale-smoothie'
group by 1
order by 1
limit 10;
```

DAILY SUM CONCLUSION

Those numbers are pretty low! A typical day has thousands in revenue but a small portion of that is coming from kale smoothies.

Let's dig deeper to find out what's going on.

MEAL SUMS

It looks like the smoothies might not be performing well, but to be sure we need to see how they're doing in the context of the other order items.
We'll look at the data several different ways, the first of which is determining what percent of revenue these smoothies represent.

1. To get the percent of revenue that each item represents, we need to know the total revenue of each item. We will later divide the per-item total with the overall total revenue.

   The following query groups and sum the products by price to get the total revenue for each item. Complete the query by passing in `sum(amount_paid)` into the `round` function and rounding to two decimal places.

```
select name, round(/**/, 2)
from order_items
```

```
group by name
order by 2 desc;
```

```
select name, round(sum(amount_paid), 2) as total_revenue
from order_items
group by name
order by 2 desc;
```

PRODUCT SUM 2

Great! We have the sum of the the products by revenue, but we still need the percent. For that, we'll need to get the total using a subquery. A *subquery* can perform complicated calculations and create filtered or aggregate tables on the fly.

Subqueries are useful when you want to perform an aggregation outside the context of the current query. This will let us calculate the overall total and per-item total at the same time.

1. Complete the denominator in the subquery, which is the total revenue from `order_items`. Use the `sum` function to query the `amount_paid` from the `order_items` table.

```
select name, round(sum(amount_paid) /
  (select /**/ from order_items) * 100.0, 2) as pct
from order_items
group by 1
order by 2 desc;
```

We now have the percent or revenue each product represents! Here `order by 2 desc` sorts by the second column (the percent) to show the products in order of their contribution to revenue.

```
select name, round(sum(amount_paid) /
  (select sum(amount_paid) from order_items) * 100.0, 2) as pct
from order_items
group by 1
order by 2 desc;
```

PRODUCT SUM CONCLUSION

As we suspected, kale smoothies are not bringing in the money. And thanks to this analysis, we found what might be a trend - several of the other low performing products are also smoothies.
Let's keep digging to find out what's going on with these smoothies.

GROUPING WITH CASE STATEMENTS

To see if our smoothie suspicion has merit, let's look at purchases by category. We can group the order items by what type of food they are, and go from there. Since our `order_items` table does not include categories already, we'll need to make some!

Previously we've been using `group by` with a column (like `order_items.name`) or a function (like `date(orders.ordered_at)`).

We can also use `group by` with expressions. In this case a `case` statement is just what we need to build our own categories. `case` statements are similar to `if/else` in other languages.
Here's the basic structure of a `case` statement:

```
case {condition}
  when {value1} then {result1}
  when {value2} then {result2}
  else {result3}
end
```

1. We'll build our own categories using a `case` statement. Complete the query below with a `case` condition of `name` that lists out each product, and decides its group.

```
select
  /**/
    when 'kale-smoothie'    then 'smoothie'
    when 'banana-smoothie'  then 'smoothie'
    when 'orange-juice'     then 'drink'
    when 'soda'             then 'drink'
    when 'blt'              then 'sandwich'
    when 'grilled-cheese'   then 'sandwich'
    when 'tikka-masala'     then 'dinner'
    when 'chicken-parm'     then 'dinner'
     else 'other'
  end as category
from order_items
order by id
limit 100;
```

Note that the `else 'other'` block catches all the products that don't meet the previous conditions.

```
select name,
  case name
    when 'kale-smoothie'    then 'smoothie'
    when 'banana-smoothie'  then 'smoothie'
    when 'orange-juice'     then 'drink'
    when 'soda'             then 'drink'
    when 'blt'              then 'sandwich'
    when 'grilled-cheese'   then 'sandwich'
    when 'tikka-masala'     then 'dinner'
    when 'chicken-parm'     then 'dinner'
     else 'other'
  end as category
from order_items
order by id
limit 10;
```

2. Complete the query by using the `category` column created by the `case` statement in our previous revenue percent calculation. Add the denominator that will `sum` the `amount_paid`.

```
select
  case name
    when 'kale-smoothie'    then 'smoothie'
    when 'banana-smoothie'  then 'smoothie'
    when 'orange-juice'     then 'drink'
    when 'soda'             then 'drink'
    when 'blt'              then 'sandwich'
    when 'grilled-cheese'   then 'sandwich'
    when 'tikka-masala'     then 'dinner'
    when 'chicken-parm'     then 'dinner'
    else 'other'
  end as category, round(1.0 * sum(amount_paid) /
    (select /**/ from order_items) * 100, 2) as pct
from order_items
group by 1
order by 2 desc;
```

Here `1.0 *` is a shortcut to ensure the database represents the percent as a decimal.

```
select
  case name
    when 'kale-smoothie'    then 'smoothie'
    when 'banana-smoothie'  then 'smoothie'
    when 'orange-juice'     then 'drink'
    when 'soda'             then 'drink'
    when 'blt'              then 'sandwich'
    when 'grilled-cheese'   then 'sandwich'
    when 'tikka-masala'     then 'dinner'
    when 'chicken-parm'     then 'dinner'
    else 'other'
  end as category, round(1.0 * sum(amount_paid) /
    (select sum(amount_paid) from order_items) * 100, 2) as pct
from order_items
group by 1
order by 2 desc;
```

PRODUCT GROUPING CONCLUSION

Ah ha! It's true that the whole smoothie category is performing poorly compared to the others. We'll certainly take this discovery to SpeedySpoon. Before we do, let's go one level deeper and figure out why.

REORDER RATES

While we do know that kale smoothies (and drinks overall) are not driving a lot of revenue, we don't know why. A big part of data analysis is implementing your own metrics to get information out of the piles of data in your database.

In our case, the reason could be that no one likes kale, but it could be something else entirely. To find out, we'll create a metric called *reorder rate* and see how that compares to the other products at SpeedySpoon.

We'll define *reorder rate* as the ratio of the total number of orders to the number of people making those orders. A higher ratio means most of the orders are reorders. A lower ratio means more of the orders are first purchases.

1. Let's calculate the reorder ratio for all of SpeedySpoon's products and take a look at the results. Counting the total orders per product is straightforward. We count the distinct `order_ids` in the `order_items` table.

   Complete the query by passing in the `distinct` keyword and the `order_id` column name into the `count` function

```
select name, /**/
from order_items
group by 1
order by 1;
```

Here's a hint on how to use the count function to count distinct columns in a table.

```
select name, count(distinct(order_id)) as orders
from order_items
group by 1
order by 1
limit 10;
```

2. Now we need the number of people making these orders. To get that information, we need to join in the `orders` table and `count` unique values in the `delivered_to` field, and sort by the `reorder_rate`.

   Complete the query below. The numerator should count the distinct `order_id`s. The denominator should count the distinct values of the `orders` table's `delivered_to` field (`orders.delivered_to`).

```
select name, round(1.0 * count(/**/) /
  count(/**/), 2) as reorder_rate
from order_items
  join orders on
    orders.id = order_items.order_id
group by 1
order by 2 desc;
```

```
select name, round(1.0 * count(distinct order_id) /
```

```
  count(distinct delivered_to), 2) as reorder_rate
from order_items
  join orders on
    orders.id = order_items.order_id
group by 1
order by 2 desc;
```

CONCLUSION

Wow! That's unexpected. While smoothies aren't making a lot of money for SpeedySpoon, they have a very high reorder rate. That means these smoothie customers are strong repeat customers.
Instead of recommending smoothies be taken off the menu, we should talk to the smoothie customers and see what they like so much about these smoothies. There could be an opportunity here to expand the product line, or get other customers as excited as these kale fanatics. Nice work!
Let's generalize what we've learned so far:

- *Data aggregation* is the grouping of data in summary form.
- *Daily Count* is the count of orders in a day.
- *Daily Revenue Count* is the revenue on orders per day.
- *Product Sum* is the total revenue of a product.
- *Subqueries* can be used to perform complicated calculations and create filtered or aggregate tables on the fly.
- *Reorder Rate* is the ratio of the total number of orders to the number of people making orders.