This is the accompanying class "notes" for the O'Reilly Online Training *Intermediate SQL for Data Analysis*.

Section I - Setup

To set up, you will need two things: SQLiteStudio and the files for this class (which you likely have already if you are reading this document).

SQLiteStudio can be downloaded from its website: https://sqlitestudio.pl/index.rvt?act=download

The files for this class can be downloaded here: https://github.com/thomasnield/oreilly_intermediate_sql_for_data

Import the "thunderbird_manufacturing.db" database file, which we will be using for almost all of the examples.

Section II - Subqueries, Unions, and Advanced Aggregations

2.1A - Scalar Subqueries

Get all orders on the last day there were orders

```
SELECT * FROM CUSTOMER_ORDER
WHERE ORDER_DATE = (SELECT MAX(ORDER_DATE) FROM CUSTOMER_ORDER)
```

2.1B - Multi-value Subqueries

Sometimes it can be helpful to leverage subqueries that return a set of values, rather than one scalar value. For instance, to query customer orders for customers in TX, we can save ourselves a join and use a subquery to get CUSTOMER_ID's that belong to customers in TX. Then we can leverage that with a WHERE and specify an IN condition:

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY

FROM CUSTOMER_ORDER

WHERE CUSTOMER_ID IN (
    SELECT CUSTOMER_ID
    FROM CUSTOMER
    WHERE STATE = 'TX'
)
```

2.2 - Aliasing Tables and Scalar Subquery Aggregation

Retrieving the average of quantity by each record's CUSTOMER_ID and PRODUCT_ID

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
(
    SELECT AVG(QUANTITY)
    FROM CUSTOMER_ORDER c2
    WHERE c2.CUSTOMER_ID = c1.CUSTOMER_ID
    AND c2.PRODUCT_ID = c1.PRODUCT_ID
) AS AVG_QUANTITY
FROM CUSTOMER_ORDER c1
```

Depending on how they are used, subqueries can be more expensive or less expensive than joins. Subqueries that generate a value for each record tend to me more expensive, like the example above.

2.3 - Derived Tables

A more efficient way to bring in averages by CUSTOMER_ID and PRODUCT_ID is by deriving a table of these averages, and joining to it.

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ORDER.CUSTOMER_ID,
ORDER_DATE,
CUSTOMER_ORDER.PRODUCT_ID,
QUANTITY,
cust_avgs.avg_qty

FROM CUSTOMER_ORDER
INNER JOIN
(
    SELECT CUSTOMER_ID,
    PRODUCT_ID,
    AVG(QUANTITY) as avg_qty
    FROM CUSTOMER_ORDER
    GROUP BY 1, 2
) cust_avgs
ON CUSTOMER_ORDER.CUSTOMER_ID = cust_avgs.CUSTOMER_ID
AND CUSTOMER_ORDER.PRODUCT_ID = cust_avgs.PRODUCT_ID
```

2.4 Common Table Expressions

You can actually create Common Table Expressions to "re-use" one or more derived tables. These can be helpful to simply provide names to derived tables, and simplify your queries greatly.

```
WITH CUST_AVGS AS (
    SELECT CUSTOMER_ID,
    PRODUCT_ID,
    AVG(QUANTITY) AS AVG_QTY
    FROM CUSTOMER_ORDER
    GROUP BY 1, 2
)

SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ORDER.CUSTOMER_ID,
ORDER_DATE,
CUSTOMER_ORDER.PRODUCT_ID,
QUANTITY,
AVG_QTY

FROM CUSTOMER_ORDER INNER JOIN CUST_AVGS

ON CUSTOMER_ORDER.CUSTOMER_ID = cust_avgs.CUSTOMER_ID
AND CUSTOMER_ORDER.PRODUCT_ID = cust_avgs.PRODUCT_ID
```

For instance, we can create two derived tables "TX_CUSTOMERS" and "TX_ORDERS" but give them names as common table expressions. Then we can proceed to use those two derived tables like this.

```
WITH TX_CUSTOMERS AS
(
SELECT * FROM CUSTOMER
WHERE STATE = 'TX'
),

TX_ORDERS AS
(
SELECT * FROM CUSTOMER_ORDER
WHERE CUSTOMER_ID IN (SELECT CUSTOMER_ID FROM TX_CUSTOMERS)
)

SELECT * FROM  TX_ORDERS INNER JOIN TX_CUSTOMERS
ON TX_ORDERS.CUSTOMER_ID = TX_CUSTOMERS.CUSTOMER_ID
```

Conditional Subqueries

```
SELECT * FROM CUSTOMER c
WHERE EXISTS (SELECT * FROM CUSTOMER_ORDER WHERE CUSTOMER_ID = c.CUSTOMER_ID)
```

2.5 - Unions

To simply append two queries (with identical fields) together, put a UNION ALL between them.

```sql
SELECT
'FEB' AS MONTH,
PRODUCT.PRODUCT_ID,
PRODUCT_NAME,
SUM(PRICE * QUANTITY) AS REV
FROM PRODUCT LEFT JOIN CUSTOMER_ORDER
ON PRODUCT.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID

WHERE ORDER_DATE BETWEEN '2017-02-01' AND '2017-02-28'
GROUP BY 1,2,3

UNION ALL

SELECT
'MAR' AS MONTH,
PRODUCT.PRODUCT_ID,
PRODUCT_NAME,
SUM(PRICE * QUANTITY) AS REV
FROM PRODUCT LEFT JOIN CUSTOMER_ORDER
ON PRODUCT.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'
GROUP BY 1,2,3
```

Using `UNION` instead of `UNION ALL` will remove duplicates, which should not be necessary in this case.

You should strive not to use unions as they often encourage bad, inefficient SQL. Strive to use CASE statements or other tools instead. In this example, it would have been better to do this:

```sql
SELECT
CASE
    WHEN ORDER_DATE BETWEEN '2017-02-01' AND '2017-02-28' THEN 'FEB'
    WHEN ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31' THEN 'MAR'
END AS MONTH,
PRODUCT.PRODUCT_ID,
PRODUCT_NAME,
SUM(PRICE * QUANTITY) AS REV
FROM PRODUCT LEFT JOIN CUSTOMER_ORDER
ON PRODUCT.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID

WHERE ORDER_DATE BETWEEN '2017-02-01' AND '2017-03-31'
GROUP BY 1,2,3
```

2.6 - GROUP CONCAT

A neat little trick you can do on some database platforms (like SQLite, MySQL, and PostgreSQL) is the `group_concat()` aggregate function. This will concatenate all values in a column as an aggregation, and can be used in conjunction with a GROUP BY like MIN, MAX, AVG, etc.

This shows a concatenated list of values of PRODUCT_ID's ordered for each ORDER_DATE.

```
SELECT ORDER_DATE,
group_concat(PRODUCT_ID) as product_ids_ordered

FROM CUSTOMER_ORDER
WHERE ORDER_DATE BETWEEN '2017-02-01' AND '2017-02-28'
GROUP BY ORDER_DATE
```

Putting the DISTINCT keyword inside of it will only concatenate the DISTINCT product ID's.

```
SELECT ORDER_DATE,
group_concat(DISTINCT PRODUCT_ID) as product_ids_ordered

FROM CUSTOMER_ORDER
WHERE ORDER_DATE BETWEEN '2017-02-01' AND '2017-02-28'
GROUP BY ORDER_DATE
```

GROUP_CONCAT is a helpful function to compress the results into a single record, in a single cell, often in a reporting context.

Note that `GROUP_CONCAT` is used in MySQL and SQLite, but is often called `STRING_AGG` on other platforms such as Oracle, PostgreSQL, and Microsoft SQL Server.

Exercise 1

Bring in all records for `CUSTOMER_ORDER`, but also bring in the total quantities ever ordered each given `PRODUCT_ID` and `CUSTOMER_ID`.

ANSWER:

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ORDER.CUSTOMER_ID,
ORDER_DATE,
CUSTOMER_ORDER.PRODUCT_ID,
QUANTITY,
sum_qty

FROM CUSTOMER_ORDER
INNER JOIN
(
    SELECT CUSTOMER_ID,
    PRODUCT_ID,
```

```
    SUM(QUANTITY) AS sum_qty
    FROM CUSTOMER_ORDER
    GROUP BY 1, 2
) total_ordered

ON CUSTOMER_ORDER.CUSTOMER_ID = total_ordered.CUSTOMER_ID
AND CUSTOMER_ORDER.PRODUCT_ID = total_ordered.PRODUCT_ID
```

Or if you choose to use a common table expression…

```
WITH total_ordered AS (
    SELECT CUSTOMER_ID,
    PRODUCT_ID,
    SUM(QUANTITY) AS sum_qty
    FROM CUSTOMER_ORDER
    GROUP BY 1, 2
)

SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ORDER.CUSTOMER_ID,
ORDER_DATE,
CUSTOMER_ORDER.PRODUCT_ID,
QUANTITY,
sum_qty

FROM CUSTOMER_ORDER INNER JOIN total_ordered

ON CUSTOMER_ORDER.CUSTOMER_ID = total_ordered.CUSTOMER_ID
AND CUSTOMER_ORDER.PRODUCT_ID = total_ordered.PRODUCT_ID
```

Section III - Regular Expressions

Regular expressions are a powerful tool for qualifying complex text patterns. Their usage extends far outside of SQL and can be found on many technology platforms including Python, R, Java, .NET, LibreOffice, Alteryx, and Tableau.

While regular expressions can be used to split and search text, we will primarily be using it to match text, much like wildcards.

The REGEXP operator is used in SQLite for matching a text to a regex pattern. Like all boolean operations, it will return a 1 for true or 0 for false. I also published an article with O'Reilly that covers regular expressions in a bit more depth here: https://www.oreilly.com/ideas/an-introduction-to-regular-expressions

Note carefully that the REGEXP operator is not implemented by default with SQLite. SQLiteStudio does implement it for you, so it will work while using SQLiteStudio. However, if you use SQLite with Python, Java, or other programming platforms you will have to implement it yourself.

3.1 - Literals

Literals are characters in a regex pattern that have no special function, and represent that character verbatim. Numbers and letters are literals. For example, The regex TX will match the string TX

```
SELECT 'TX' REGEXP 'TX' --true
```

Some characters, as we have seen, have special functionality in a regex. If you want to use these characters as literals, sometimes you have to escape them with a preceding \. These characters include:

[\^$.|?*+()

So to qualify a U.S. currency amount, you will need to escape the dollar sign $ and the decimal place .

```
SELECT '$181.12' REGEXP '\$181\.12' -- true
```

3.1 - Qualifying Alphabetic and Numeric Ranges

A range is a valid set of values for a single character. For instance [A-Z] qualifies one uppercase alphabetic value for the range of letters A thru Z, so [A-Z][A-Z] would qualify two uppercase alphabetic text values. This would match the text string TX.

```
SELECT 'TX' REGEXP '[A-Z][A-Z]' --true
SELECT '45' REGEXP '[A-Z][A-Z]' --false
SELECT 'T2' REGEXP '[A-Z][0-3]' --true
SELECT 'T9' REGEXP '[A-Z0-9][A-Z0-9]' --true
```

TX would not match [A-Z][A-Z][A-Z] though because it is not three characters.

```
SELECT 'TX' REGEXP '[A-Z][A-Z][A-Z]' --false
SELECT 'ASU' REGEXP '[A-Z][A-Z][A-Z]' --true
```

We can also specify certain characters, and they don't necessarily have to be ranges:

```
SELECT 'A6' REGEXP '[ATUX][469]' --true
SELECT 'B8' REGEXP '[ATUX][469]' --false
```

Conversely, we can negate a set of characters by starting the range with ^:

```
SELECT 'A6' REGEXP '[^ATUX][^469]' --false
SELECT 'B8' REGEXP '[^ATUX][^469]' --true
```

3.2 - Anchoring

If you don't want partial matches but rather full matches, you have to anchor the beginning and end of the String with ^ and $ respectively.

For instance, [A-Z][A-Z] would qualify with SMU. This is because it found two alphabetic characters within those three characters.

```sql
SELECT 'SMU' REGEXP '[A-Z][A-Z]' --true
```

If you don't want that, you will need to qualify start and end anchors, which effectively demands a full match when both are used:

```sql
SELECT 'SMU' REGEXP '^[A-Z][A-Z]$' --false
```

You can also anchor to just the beginning or end of the string to check, for instance, if a string starts with a number followed by an alphabetic character:

```sql
SELECT '9FN' REGEXP '^[0-9][A-Z]' --true
SELECT 'RFX' REGEXP '^[0-9][A-Z]' --false
```

3.3 - Repeaters

Sometimes we simply want to qualify a repeated pattern in our regular expression. For example, this is redundant:

```sql
SELECT 'ASU' REGEXP '^[A-Z][A-Z][A-Z]$' --true
```

We can instead explicitly identify in curly brackets we want to repeat that alphabetic character 3 times, by following it with a {3}.

```sql
SELECT 'ASU' REGEXP '^[A-Z]{3}$' --true
```

We can also specify a min and max number of repetitions, such as a minimum of 2 but max of 3.

```sql
SELECT 'ASU' REGEXP '^[A-Z]{2,3}$' --true
SELECT 'TX' REGEXP '^[A-Z]{2,3}$' --true
```

Leaving the second argument blank will result in only requiring a minimum of repetitions:

```sql
SELECT 'A' REGEXP '^[A-Z]{2,}$' --false
SELECT 'ASDIKJFSKJJNXVNJGTHEWIROQWERKJTX' REGEXP '^[A-Z]{2,}$' --true
```

To allow 1 or more repetitions, use the +. This will qualify with 1 or more alphanumeric characters.

```sql
SELECT 'ASDFJSKJ4892KSFJJ2843KJSNBKW' REGEXP '^[A-Z0-9]+$' --true
SELECT 'SDFJSDKJF/&SSDKJ$#SDFKSDFKJ' REGEXP '^[A-Z0-9]+$' --false
```

To allow 0 or more repetitions, use the *

```sql
SELECT 'ASDFJSKJ4892KSFJJ2843KJSNBKW' REGEXP '^[A-Z0-9]*' --true
SELECT '' REGEXP '^[A-Z0-9]*' --true
```

To allow 0 or 1 repetitions (an optional character), follow the item with a ?. This will allow two characters to be preceded with a number, but it doesn't have to:

```sql
SELECT '9FX' REGEXP '^[0-9]?[A-Z]{2}$' --true
SELECT 'AX' REGEXP '^[0-9]?[A-Z]{2}$' --true
```

You can use several repeaters for different clauses in a regex. Below, we qualify a string of alphabetic characters, a dash - followed by a string of numbers, and then another - with a string of alphabetic characters.

```sql
SELECT 'ASJSDFH-32423522-HUETHNB' REGEXP '^[A-Z]+-[0-9]+-[A-Z]+$' --true
```

3.4 Wildcards

A dot `.` represents any character, even whitespaces.

```sql
SELECT 'A-3' REGEXP '...' --true
```

You can also use it with repeaters to create broad wildcards for any number of characters.

```sql
SELECT 'A-3' REGEXP '.{3}' --true
SELECT 'A-3' REGEXP '.+' --true
SELECT 'A-3' REGEXP '.*' --true
```

`.*` is a common way to express qualifying any text.

3.5 Alternation and Grouping

You can group up parts of a regular expression using rounded paranthesis ( ), often to put a repeater on that entire group. For example, we can make the entire decimal part of a dollar amount optional:

```sql
SELECT '181.12' REGEXP '^[0-9]+(\.[0-9]{2})?$' --true
SELECT '181' REGEXP '^[0-9]+(\.[0-9]{2})?$' --true
``
```

```
We can also qualify a string of letters, a slash `/`, and a string of
numbers, but qualify any number of repetitions of this entire pattern:

```sql
SELECT 'WHISKY/23482374/ZULU/23423234/FOXTROT/6453' REGEXP '^([A-Z]+/[0-
9]+/?)+$' --true
```

The pipe | operator functions as an alternator operator, or effectively an OR. It allows you to qualify any number of regular expressions where at least one of them must be true:

```sql
SELECT 'ALPHA' REGEXP '^(FOXTROT|ZULU|ALPHA|TANGO)$' --true
```

3.7 Using Regular Expressions in queries

Find all customers with a 3-4 digit street number. Note the literal space before the wildcard `.*`:

```
SELECT * FROM CUSTOMER
WHERE ADDRESS REGEXP '^[0-9]{3,4} .*$'
```

EXERCISE

Find all customers with an address ending in "Blvd" or "St":

```
SELECT * FROM CUSTOMER
WHERE ADDRESS REGEXP '.*(Blvd|St)$'
```

Section IV - Advanced Joins and Temporary Tables

4.1 Inner Join Review

Using an INNER JOIN, you can view CUSTOMER and PRODUCT information with each CUSTOMER_ORDER.

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_NAME,
ORDER_DATE,
PRODUCT_ID,
PRODUCT_NAME,
PRODUCT_GROUP
QUANTITY,
PRICE

FROM CUSTOMER_ORDER

INNER JOIN CUSTOMER
ON CUSTOMER_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID

INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID
```

4.2 Left Join Review

Total quantity sold of all products, null means no products were sold that day.

```
SELECT PRODUCT_ID,
PRODUCT_NAME,
SUM(QUANTITY) as total_quantity

FROM PRODUCT LEFT JOIN CUSTOMER_ORDER

ON PRODUCT.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID
AND ORDER_DATE = '2017-03-01'

GROUP BY 1, 2
```

4.3 Creating a Volatile Table

Herei show to create a volatile/temporary table of discount rules. This table will dispose at the end of each session. It is no different than a standard `CREATE TABLE` statement other than the `TEMP` keyword.

```sql
CREATE TEMP TABLE DISCOUNT (
    CUSTOMER_ID_REGEX    VARCHAR (20) NOT NULL DEFAULT ('.*'),
    PRODUCT_ID_REGEX     VARCHAR (20) NOT NULL DEFAULT ('.*'),
    PRODUCT_GROUP_REGEX VARCHAR (30) NOT NULL DEFAULT ('.*'),
    STATE_REGEX          VARCHAR (30) NOT NULL DEFAULT ('.*'),
    DISCOUNT_RATE        DOUBLE       NOT NULL
);

INSERT INTO DISCOUNT (STATE_REGEX, DISCOUNT_RATE) VALUES ('LA|OK', 0.20);
INSERT INTO DISCOUNT (PRODUCT_GROUP_REGEX, STATE_REGEX, DISCOUNT_RATE) VALUES
('BETA|GAMMA','TX', 0.10);
INSERT INTO DISCOUNT (PRODUCT_ID_REGEX, CUSTOMER_ID_REGEX, DISCOUNT_RATE)
VALUES ('^[379]$', '^(1|6|12)$', 0.30);
```

Note you can also create a temporary (or permanent) table from a SELECT query. This is helpful to persist expensive query results and reuse it multiple times during a session. SQLite is a bit more convoluted to do this than other platforms:

```sql
CREATE TEMP TABLE ORDER_TOTALS_BY_DATE AS
WITH ORDER_TOTALS_BY_DATE AS (
    SELECT ORDER_DATE,
    SUM(QUANTITY) AS TOTAL_QUANTITY
    FROM CUSTOMER_ORDER
    GROUP BY 1
)
SELECT * FROM ORDER_TOTALS_BY_DATE
```

4.4 Joining with Regular Expressions

Left-joining to the temporary table and qualifying on the regular expressions for each respective field allows us to apply the discounts to each `CUSTOMER_ORDER` as specified.

```sql
SELECT CUSTOMER_ORDER.*,
PRICE,
DISCOUNT_RATE,
PRICE * (1 - DISCOUNT_RATE) AS DISCOUNTED_PRICE

FROM CUSTOMER_ORDER
INNER JOIN CUSTOMER
ON CUSTOMER_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID

INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID
```

```
LEFT JOIN DISCOUNT
ON CUSTOMER_ORDER.CUSTOMER_ID REGEXP DISCOUNT.CUSTOMER_ID_REGEX
AND CUSTOMER_ORDER.PRODUCT_ID REGEXP DISCOUNT.PRODUCT_ID_REGEX
AND PRODUCT.PRODUCT_GROUP REGEXP DISCOUNT.PRODUCT_GROUP_REGEX
AND CUSTOMER.STATE REGEXP DISCOUNT.STATE_REGEX

WHERE ORDER_DATE BETWEEN '2017-03-26' AND '2017-03-31'
```

If you expect records to possibly get multiple discounts, then sum the discounts and GROUP BY everything else:

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_NAME,
STATE,
ORDER_DATE,
CUSTOMER_ORDER.PRODUCT_ID,
PRODUCT_NAME,
PRODUCT_GROUP
QUANTITY,
PRICE,
SUM(DISCOUNT_RATE) as TOTAL_DISCOUNT_RATE,
PRICE * (1 - SUM(DISCOUNT_RATE)) AS DISCOUNTED_PRICE

FROM CUSTOMER_ORDER
INNER JOIN CUSTOMER
ON CUSTOMER_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID

INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = CUSTOMER_ORDER.PRODUCT_ID

LEFT JOIN DISCOUNT
ON CUSTOMER_ORDER.CUSTOMER_ID REGEXP DISCOUNT.CUSTOMER_ID_REGEX
AND CUSTOMER_ORDER.PRODUCT_ID REGEXP DISCOUNT.PRODUCT_ID_REGEX
AND PRODUCT.PRODUCT_GROUP REGEXP DISCOUNT.PRODUCT_GROUP_REGEX
AND CUSTOMER.STATE REGEXP DISCOUNT.STATE_REGEX

WHERE ORDER_DATE BETWEEN '2017-03-26' AND '2017-03-31'

GROUP BY 1,2,3,4,5,6,7,8
```

4.5A Self Joins

We can join a table to itself by invoking it twice with two aliases. This can be useful, for example, to look up the previous day's order quantity (if any) for a given CUSTOMER_ID and PRODUCT_ID:

```
SELECT o1.CUSTOMER_ORDER_ID,
o1.CUSTOMER_ID,
o1.PRODUCT_ID,
```

```
        o1.ORDER_DATE,
        o1.QUANTITY,
        o2.QUANTITY AS PREV_DAY_QUANTITY

FROM CUSTOMER_ORDER o1
LEFT JOIN CUSTOMER_ORDER o2

ON o1.CUSTOMER_ID = o2.CUSTOMER_ID
AND o1.PRODUCT_ID = o2.PRODUCT_ID
AND o2.ORDER_DATE = date(o1.ORDER_DATE, '-1 day')

WHERE o1.ORDER_DATE BETWEEN '2017-03-05' AND '2017-03-11'
```

Note if you want to get the previous quantity ordered for that record's given CUSTOMER_ID and PRODUCT_ID, even if it wasn't strictly the day before, you can use a subquery instead that qualifies previous dates and orders them descending. Then you can use LIMIT 1 to grab the most recent at the top.

```
SELECT ORDER_DATE,
PRODUCT_ID,
CUSTOMER_ID,
QUANTITY,
(
    SELECT QUANTITY
    FROM CUSTOMER_ORDER c2
    WHERE c1.ORDER_DATE > c2.ORDER_DATE
    AND c1.PRODUCT_ID = c2.PRODUCT_ID
    AND c1.CUSTOMER_ID = c2.CUSTOMER_ID
    ORDER BY ORDER_DATE DESC
    LIMIT 1
) as PREV_QTY
FROM CUSTOMER_ORDER c1
```

4.5B Recursive Self Joins

At some point of your career, you may encounter a table that is inherently designed to be self-joined. For instance, run this query:

```
SELECT * FROM EMPLOYEE
```

This is a table containing employee information, including their manager via a MANAGER_ID field. Here is a sample of the results below.

ID	FIRST_NAME	LAST_NAME	TITLE	DEPARTMENT	MANAGER_ID
13	Pembroke	Killgus	Accountant I	Accounting	10
14	Harper	Argontt	Director	Operations	3
15	Fabio	Treversh	Manager	Operations	14
16	Gerard	Morforth	Analyst	Operations	15

17	Stephanus	Palatino	Senior Analyst	Operations	15
18	Jennilee	Withers	Analyst	Operations	15
19	Desdemona	Farmar	Business Consultant	Operations	15
20	Ashlin	Creamen	Manager	Operations	14
21	Daniel	Licquorish	Analyst	Operations	20

This MANAGER_ID points to another EMPLOYEE record. If you want to bring in Daniel and his superior's information, this isn't hard to do with a self join.

```sql
SELECT e1.FIRST_NAME,
e1.LAST_NAME,
e1.TITLE,
e2.FIRST_NAME AS MANAGER_FIRST_NAME,
e2.LAST_NAME AS MANAGER_LAST_NAME

FROM EMPLOYEE e1 INNER JOIN EMPLOYEE e2
ON e1.MANAGER_ID = e2.ID

WHERE e1.FIRST_NAME = 'Daniel'
```

FIRST_NAME	LAST_NAME	TITLE	MANAGER_FIRST_NAME	MANAGER_LAST_NAME
Daniel	Licquorish	Analyst	Ashlin	Creamen

But what if you wanted to display the entire hierarchy above Daniel? Well shoot, this is hard because now you have to do several self joins to daisy-chain your way to the top. What makes this even harder is you don't know how many self joins you will need to do. For cases like this, it can be helpful to leverage recursive queries.

A recursion is a special type of common table expression (CTE). Typically, you "seed" a starting value and then use UNION or UNION ALL to append the results of a query that uses each "seed", and the result becomes the next seed.

In this case, we will use a RECURSIVE common table expression to seed Daniel's ID, and then append each MANAGER_ID of each EMPLOYEE_ID that matches the seed. This will give a set of ID's for employees hierarchical to Daniel. We can then use these ID's to navigate Daniel's hierarchy via JOINS, IN, or other SQL operators.

```sql
-- generates a list of employee ID's hierarchical to Daniel

WITH RECURSIVE hierarchy_of_daniel(x) AS (
 SELECT 21 -- start with Daniel's ID
 UNION ALL -- append each manager ID recursively
 SELECT MANAGER_ID
 FROM hierarchy_of_daniel INNER JOIN EMPLOYEE
 ON EMPLOYEE.ID = hierarchy_of_daniel.x -- employee ID must equal previous
recursion
)
```

```sql
SELECT * FROM EMPLOYEE
WHERE ID IN hierarchy_of_daniel;
```

Recursive queries are a bit tricky to get right, but practice them if you deal frequently with hierarchical records. You will likely use them with a specific part of the hierarchy in focus (e.g. Daniel's superiors). It's harder to show the hierarchy for everyone at once, but there are ways. For instance, you can put a RECURSIVE operation in a subquery and use GROUP_CONCAT.

```sql
SELECT e1.* ,

(
    WITH RECURSIVE hierarchy_of(x) AS (
     SELECT e1.ID
     UNION ALL -- append each manager ID recursively
     SELECT MANAGER_ID
     FROM hierarchy_of INNER JOIN EMPLOYEE
     ON EMPLOYEE.ID = hierarchy_of.x -- employee ID must equal previous
recursion
      )

    SELECT GROUP_CONCAT(ID) FROM EMPLOYEE e2
    WHERE ID IN hierarchy_of
) AS HIERARCHY_IDS

FROM EMPLOYEE e1
```

ID	FIRST_NAME	LAST_NAME	TITLE	DEPARTMENT	MANAGER_ID	HIERARCHY_IDS
14	Harper	Argontt	Director	Operations	3	1,3,14
15	Fabio	Treversh	Manager	Operations	14	1,3,14,15
16	Gerard	Morforth	Analyst	Operations	15	1,3,14,15,16
17	Stephanus	Palatino	Senior Analyst	Operations	15	1,3,14,15,17
18	Jennilee	Withers	Analyst	Operations	15	1,3,14,15,18
19	Desdemona	Farmar	Business Consultant	Operations	15	1,3,14,15,19
20	Ashlin	Creamen	Manager	Operations	14	1,3,14,20
21	Daniel	Licquorish	Analyst	Operations	20	1,3,14,20,21

2 2	Quill	Pinder	Senior Analyst	Operations	20	1,3,14,20,22
2 3	Maybelle	Freiburger	Business Consultant	Operations	20	1,3,14,20,23
2 4	Angelique	Havis	Business Consultant	Operations	20	1,3,14,20,24
2 5	Lyn	Geale	Director	Technology	4	1,4,25
2 6	Tammy	Eakly	Manager	Help Desk	25	1,4,25,26
2 7	Junie	Blanque	Technician I	Help Desk	26	1,4,25,26,27

Note recursive queries also can be used to improvise a set of consecutive values without creating a table. For instance, we can generate a set of consecutive integers. Here is how you create a set of integers from 1 to 1000.

```
WITH RECURSIVE my_integers(x) AS (
    SELECT 1
        UNION ALL
    SELECT x + 1
    FROM my_integers
    WHERE x < 1000
)
SELECT * FROM my_integers
```

Generating integers can also be be helpful to "repeat-and-modify" records in a given table. For example, if we have a table of air travel bookings where each booking can have "x" number of passengers (such as 3 passengers).

BOOKING_ID	BOOKED_EMPLOYEE_ID	DEPARTURE_DATE	ORIGIN	DESTINATION	FARE_PRICE	NUM_OF_PASSENGERS	RETURN_BOOKING_ID
1	6	2017-03-01	DFW	ORD	170	2	2
2	6	2017-03-04	ORD	DFW	160	2	
3	19	2017-03-21	DFW	JFK	210	3	4
4	19	2017-03-24	JFK	DFW	220	3	
5	1	2017-03-26	DFW	LAX	180	1	6

6	1	2017-03-27	LAX	DFW	190	1		
7	5	2017-03-27	DFW	ORD	210	2		8
8	5	2017-03-29	ORD	DFW	190	2		
9	9	2017-03-28	DFW	SFO	220	3		10
10	9	2017-03-28	SFO	DFW	230	3		
11	31	2017-04-01	DFW	LAX	190	1		12
12	31	2017-04-05	LAX	DFW	180	1		

We can break up each booking into individual bookings for each passenger (e.g create 3 records off of a booking with 3 passengers).

```
WITH RECURSIVE repeat_helper(x) AS (
    SELECT 1
        UNION ALL
    SELECT x + 1
    FROM repeat_helper
    WHERE x < 1000
)

SELECT BOOKING_ID,
BOOKED_EMPLOYEE_ID,
DEPARTURE_DATE,
ORIGIN,
DESTINATION,
FARE_PRICE,
repeat_helper.x AS PASSENGER_NUMBER
FROM EMPLOYEE_AIR_TRAVEL CROSS JOIN repeat_helper
ON repeat_helper.x <= NUM_OF_PASSENGERS
```

BOOKING_ID	BOOKED_EMPLOYEE_ID	DEPARTURE_DATE	ORIGIN	DESTINATION	FARE_PRICE	PASSENGER_NUMBER
1	6	2017-03-01	DFW	ORD	170	1
1	6	2017-03-01	DFW	ORD	170	2
2	6	2017-03-04	ORD	DFW	160	1
2	6	2017-03-04	ORD	DFW	160	2
3	19	2017-03-21	DFW	JFK	210	1
3	19	2017-03-21	DFW	JFK	210	2

3	19	2017-03-21	DFW	JFK	210	3
4	19	2017-03-24	JFK	DFW	220	1
4	19	2017-03-24	JFK	DFW	220	2
4	19	2017-03-24	JFK	DFW	220	3
5	1	2017-03-26	DFW	LAX	180	1
6	1	2017-03-27	LAX	DFW	190	1
7	5	2017-03-27	DFW	ORD	210	1

You can also use some clever CASE expression logic with an integer generater to find total costs of sending employees to each airport.

```
WITH RECURSIVE repeat_helper(x) AS (
    SELECT 1
        UNION ALL
    SELECT x + 1
    FROM repeat_helper
    WHERE x < 1000
)

SELECT
CASE WHEN repeat_helper.x == 1 THEN ORIGIN ELSE DESTINATION END AS AIRPORT,
SUM(FARE_PRICE * NUM_OF_PASSENGERS) AS AIRPORT_REVENUE
FROM EMPLOYEE_AIR_TRAVEL CROSS JOIN repeat_helper
ON repeat_helper.x <= 2
GROUP BY AIRPORT
```

AIRPORT	AIRPORT_REVENUE
DFW	4840
JFK	1290
LAX	740
ORD	1460
SFO	1350

You can apply the same concept to generate a set of chronological dates. This recursive query will generate all dates from today to '2030-12-31':

```
WITH RECURSIVE my_dates(x) AS (
    SELECT date('now')
        UNION ALL
    SELECT date(x, '+1 day')
    FROM my_dates
    WHERE x < '2030-12-31'
)
SELECT * FROM my_dates
```

4.6 Cross Joins

Sometimes it can be helpful to generate a "cartesian product", or every possible combination between two or more data sets using a CROSS JOIN. This is often done to generate a data set that fills in gaps for another query. Not every calendar date has orders, nor does every order date have an entry for every product, as shown in this query:

```sql
SELECT ORDER_DATE,
PRODUCT_ID,
SUM(QUANTITY) as TOTAL_QTY

FROM CUSTOMER_ORDER

GROUP BY 1, 2
```

We should use a cross join to resolve this problem. For instance, we can leverage a CROSS JOIN query to generate every possible combination of PRODUCT_ID and CUSTOMER_ID.

```sql
SELECT
CUSTOMER_ID,
PRODUCT_ID
FROM CUSTOMER
CROSS JOIN PRODUCT
```

In this case we should bring in CALENDAR_DATE and cross join it with PRODUCT_ID to get every possible combination of calendar date and product. Note the CALENDAR_DATE comes from the CALENDAR table, which acts as a simple list of consecutive calendar dates. Note we could also have used a recursive query, as shown in the previous example, to generate the dates. We'll stick with a simple table instead for now in case you are not comfortable with recursion yet. We should only filter the calendar to a date range of interest, like 2017-01-01 and 2017-03-31.

```sql
SELECT
CALENDAR_DATE,
PRODUCT_ID
FROM PRODUCT
CROSS JOIN CALENDAR
WHERE CALENDAR_DATE BETWEEN '2017-01-01' and '2017-03-31'
```

Then we can LEFT JOIN to our previous query to get every product quantity sold by calendar date, even if there were no orders that day:

```sql
SELECT CALENDAR_DATE,
all_combos.PRODUCT_ID,
TOTAL_QTY

FROM
(
  SELECT
  CALENDAR_DATE,
```

```sql
    PRODUCT_ID
    FROM PRODUCT
    CROSS JOIN CALENDAR
    WHERE CALENDAR_DATE BETWEEN '2017-01-01' and '2017-03-31'
) all_combos

LEFT JOIN
(
    SELECT ORDER_DATE,
    PRODUCT_ID,
    SUM(QUANTITY) as TOTAL_QTY

    FROM CUSTOMER_ORDER

    GROUP BY 1, 2
) totals

ON all_combos.CALENDAR_DATE = totals.ORDER_DATE
AND all_combos.PRODUCT_ID = totals.PRODUCT_ID

ORDER BY CALENDAR_DATE, all_combos.PRODUCT_ID
```

Note you can also use common table expressions:

```sql
WITH all_combos AS (
    SELECT
    CALENDAR_DATE,
    PRODUCT_ID
    FROM PRODUCT
    CROSS JOIN CALENDAR
    WHERE CALENDAR_DATE BETWEEN '2017-01-01' and '2017-03-31'
),

totals AS (
    SELECT ORDER_DATE,
    PRODUCT_ID,
    SUM(QUANTITY) as TOTAL_QTY

    FROM CUSTOMER_ORDER

    GROUP BY 1, 2
)


SELECT CALENDAR_DATE,
all_combos.PRODUCT_ID,
TOTAL_QTY

FROM all_combos LEFT JOIN totals
```

```
ON all_combos.CALENDAR_DATE = totals.ORDER_DATE
AND all_combos.PRODUCT_ID = totals.PRODUCT_ID

ORDER BY CALENDAR_DATE, all_combos.PRODUCT_ID
```

4.7 Comparative Joins

Note also you can use comparison operators in joins. For instance, we can self-join to create rolling quantity totals and generate a cartesian product on previous dates to the current order, and then sum those quantities. It is much easier to use windowing functions for this purpose though, which is covered in the next section.

```
SELECT c1.ORDER_DATE,
c1.PRODUCT_ID,
c1.CUSTOMER_ID,
c1.QUANTITY,
SUM(c2.QUANTITY) as ROLLING_QTY

FROM CUSTOMER_ORDER c1 INNER JOIN CUSTOMER_ORDER c2
ON c1.PRODUCT_ID = c2.PRODUCT_ID
AND c1.CUSTOMER_ID = c2.CUSTOMER_ID
AND c1.ORDER_DATE >= c2.ORDER_DATE

GROUP BY 1, 2, 3, 4
```

Exercise 4

For every CALENDAR_DATE and CUSTOMER_ID, show the total QUANTITY ordered for the date range of 2017-01-01 to 2017-03-31:

ANSWER:

```
SELECT CALENDAR_DATE,
all_combos.CUSTOMER_ID,
coalesce(TOTAL_QTY, 0) AS TOTAL_QTY

FROM
(
  SELECT
  CALENDAR_DATE,
  CUSTOMER_ID
  FROM CUSTOMER
  CROSS JOIN CALENDAR
  WHERE CALENDAR_DATE BETWEEN '2017-01-01' and '2017-03-31'
) all_combos

LEFT JOIN
(
  SELECT ORDER_DATE,
```

```sql
  CUSTOMER_ID,
  SUM(QUANTITY) as TOTAL_QTY

  FROM CUSTOMER_ORDER

  GROUP BY 1, 2
) totals

ON all_combos.CALENDAR_DATE = totals.ORDER_DATE
AND all_combos.CUSTOMER_ID = totals.CUSTOMER_ID

ORDER BY CALENDAR_DATE, all_combos.CUSTOMER_ID
```

Using Common Table Expressions:

```sql
WITH all_combos AS (
  SELECT
  CALENDAR_DATE,
  CUSTOMER_ID
  FROM CUSTOMER
  CROSS JOIN CALENDAR
  WHERE CALENDAR_DATE BETWEEN '2017-01-01' and '2017-03-31'
),

totals AS (
  SELECT ORDER_DATE,
  CUSTOMER_ID,
  SUM(QUANTITY) as TOTAL_QTY

  FROM CUSTOMER_ORDER

  GROUP BY 1, 2
)

SELECT CALENDAR_DATE,
all_combos.CUSTOMER_ID,
coalesce(TOTAL_QTY, 0) AS TOTAL_QTY

from all_combos LEFT JOIN totals

ON all_combos.CALENDAR_DATE = totals.ORDER_DATE
AND all_combos.CUSTOMER_ID = totals.CUSTOMER_ID

ORDER BY CALENDAR_DATE, all_combos.CUSTOMER_ID
```

Section V - Windowing

Windowing functions allow you to greater contextual aggregations in ways much more flexible than GROUP BY. Many major database platforms support windowing functions.

5.1 PARTITION BY

Sometimes it can be helpful to create a contextual aggregation for each record in a query. Windowing functions can make this much easier and save us a lot of subquery work.

For instance, it may be helpful to not only get each CUSTOMER_ORDER for the month of MARCH, but also the average quantity that customer purchased for that PRODUCT_ID. We can do that with an OVER PARTITION BY combined with the AVG() function. We have done this operation with derived tables and common table expressions in the past, but now we can do it in a one-line window function!

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
AVG(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
AVG_PRODUCT_QTY_ORDERED

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

Here is the maximum quantity for shared customers and products:

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
MAX(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
MAX_PRODUCT_QTY_ORDERED

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

Each MAX_PRODUCT_QTY_ORDERED will only be the minimum QUANTITY of that given record's PRDOUCT_ID and CUSTOMER_ID. The WHERE will also filter that scope to only within MARCH.

You can have multiple windowed fields in a query. Below, we get a MIN, MAX, and AVG for that given window.

```sql
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
MIN(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
MIN_PRODUCT_QTY_ORDERED,
MAX(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
MAX_PRODUCT_QTY_ORDERED,
AVG(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
AVG_PRODUCT_QTY_ORDERED

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

You can also mix and match scopes which is difficult to do with derived tables.

```sql
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
MIN(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID) as
MIN_PRODUCT_CUSTOMER_QTY_ORDERED,
MIN(QUANTITY) OVER(PARTITION BY PRODUCT_ID) as MIN_PRODUCT_QTY_ORDERED,
MIN(QUANTITY) OVER(PARTITION BY CUSTOMER_ID) as MIN_CUSTOMER_QTY_ORDERED


FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'
```

When you are declaring your window redundantly, you can reuse it using a WINDOW declaration, which goes between the WHERE and the ORDER BY.

```sql
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
MIN(QUANTITY) OVER(w) as MIN_PRODUCT_QTY_ORDERED,
MAX(QUANTITY) OVER(w) as MAX_PRODUCT_QTY_ORDERED,
AVG(QUANTITY) OVER(w) as AVG_PRODUCT_QTY_ORDERED
```

```
FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

WINDOW w AS (PARTITION BY PRODUCT_ID, CUSTOMER_ID)

ORDER BY CUSTOMER_ORDER_ID
```

5.2 ORDER BY

You can also use an `ORDER BY` in your window to only consider values that comparatively come before that record.

5.2A USING ORDER BY

For instance, you can get a ROLLING_TOTAL of the QUANTITY by ordering by the ORDER_DATE.

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
SUM(QUANTITY) OVER (ORDER BY ORDER_DATE) AS ROLLING_QUANTITY

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

Note you can precede the `ORDER BY` clause with a `DESC` keyword to window in the opposite direction.

5.2B Ordering and Bounds

Above, notice our example output has the same rolling total for all records on a given date. This is because the ORDER BY in a window function by default does a logical boundary, which in this case is the `ORDER_DATE`. This means it is rolling up everything on that `ORDER_DATE` and previous to it. A side effect is all records with the same `ORDER_DATE` are going to get the same rolling total.

This is the default behavior our query did previously:

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
```

```
SUM(QUANTITY) OVER (ORDER BY ORDER_DATE RANGE BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS ROLLING_QUANTITY

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

If you want to incrementally roll the quantity by each row's physical order (not logical order by the entire ORDER_DATE), you can use ROWS BETWEEN instead of RANGE BETWEEN.

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
SUM(QUANTITY) OVER (ORDER BY ORDER_DATE ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS ROLLING_QUANTITY

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

Note the AND CURRENT ROW is a default, so you can shorthand it like this:

```
SUM(QUANTITY) OVER (ORDER BY ORDER_DATE ROWS UNBOUNDED PRECEDING) AS
ROLLING_QUANTITY
```

In this particular example, you could have avoided using a physical boundary by specifying your window with an ORDER BY CUSTOMER_ORDER_ID. But we covered the previous strategy anyway to see how to execute physical boundaries. Here is an excellent overview of windowing functions and bounds: http://mysqlserverteam.com/mysql-8-0-2-introducing-window-functions/

5.3 Mixing PARTITION BY / ORDER BY

We can combine the PARTITION BY / ORDER BY to create rolling aggregations partitioned on certain fields.

5.3A Simple MAX over PARITION and ORDER BY

For example, for each record we can get the max quantity ordered up to that date

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
MAX(QUANTITY) OVER(PARTITION BY PRODUCT_ID, CUSTOMER_ID ORDER BY ORDER_DATE)
```

```
as MAX_TO_DATE_PRODUCT_QTY

FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

5.3B Rolling Total Quantity by PRODUCT_ID, with physical boundary
```
SELECT CUSTOMER_ORDER_ID,
ORDER_DATE,
CUSTOMER_ID,
PRODUCT_ID,
QUANTITY,
SUM(QUANTITY) OVER(PARTITION BY PRODUCT_ID ORDER BY ORDER_DATE ROWS UNBOUNDED
PRECEDING) as total_qty_for_customer_and_product

FROM CUSTOMER_ORDER
WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'
```

You need to be very careful mixing PARITITION BY with an ORDER BY that uses a physical boundary! If you sort the results, it can get confusing very quickly because you lose that physical ordered context.

```
SELECT CUSTOMER_ORDER_ID,
ORDER_DATE,
CUSTOMER_ID,
PRODUCT_ID,
QUANTITY,
SUM(QUANTITY) OVER(PARTITION BY PRODUCT_ID ORDER BY ORDER_DATE) as
total_qty_for_product

FROM CUSTOMER_ORDER
WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY ORDER_DATE
```

5.4 Rolling Windows

You can also use movable windows to create moving aggregations. For instance, you can create a six-row rolling average (3 rows before, 3 rows after).

```
SELECT CUSTOMER_ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
PRODUCT_ID,
QUANTITY,
AVG(QUANTITY) OVER (ORDER BY ORDER_DATE ROWS BETWEEN 3 PRECEDING AND 3
FOLLOWING)  AS ROLLING_AVG
```

```sql
FROM CUSTOMER_ORDER

WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

EXERCISE

For the month of March, bring in the rolling sum of quantity ordered (up to each
ORDER_DATE) by CUSTOMER_ID and PRODUCT_ID.

```sql
SELECT CUSTOMER_ORDER_ID,
ORDER_DATE,
CUSTOMER_ID,
PRODUCT_ID,
QUANTITY,
SUM(QUANTITY) OVER(PARTITION BY CUSTOMER_ID, PRODUCT_ID ORDER BY ORDER_DATE)
as total_qty_for_customer_and_product

FROM CUSTOMER_ORDER
WHERE ORDER_DATE BETWEEN '2017-03-01' AND '2017-03-31'

ORDER BY CUSTOMER_ORDER_ID
```

Section VI - SQL with Python, R, and Java

6.1A Using SQL with Python

When doing SQL with Python, you want to use SQLAlchemy. Below, we query and loop
through the CUSTOMER table which is returned as an iteration of Tuples:

```python
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///thunderbird_manufacturing.db')
conn = engine.connect()

stmt = text("SELECT * FROM CUSTOMER")
results = conn.execute(stmt)

for r in results:
    print(r)
```

6.1B Using SQL with Python

You can package up interactions with a database into helper functions. Below, we create a
function called get_all_customers() which returns the results as a List of tuples:

```
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///thunderbird_manufacturing.db')
conn = engine.connect()

def get_all_customers():
    stmt = text("SELECT * FROM CUSTOMER")
    return list(conn.execute(stmt))


print(get_all_customers())
```

6.1C Using SQL with Python

If you want to pass parameters to a query, mind to not insert parameters directly so you don't accidentally introduce SQL injection. Below, we create a helper function that retrieves a customer for a given ID from a database.

```
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///thunderbird_manufacturing.db')
conn = engine.connect()

def get_all_customers():
    stmt = text("SELECT * FROM CUSTOMER")
    return list(conn.execute(stmt))


def customer_for_id(customer_id):
    stmt = text("SELECT * FROM CUSTOMER WHERE CUSTOMER_ID = :id")
    return conn.execute(stmt, id=customer_id).first()


print(customer_for_id(3))
```

6.1D Using SQL with Pandas

Using SQL with Pandas is pretty straightforward. Use Pandas' `read_sql_query()` function to query from a `connection` object and turn it into a `DataFrame`.

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('thunderbird_manufacturing.db')
data_frame_from_sql = pd.read_sql_query("SELECT * FROM CUSTOMER", conn)
print(data_frame_from_sql)
```

6.2 Using SQL with R

Here is how to run a SQL query in R, and save the results to a matrix.

```r
setwd('c:\\my_folder')

library(DBI)
library(RSQLite)

db <- dbConnect(SQLite(), dbname='thunderbird_manufacturing.db')

my_query <- dbSendQuery(db, "SELECT * FROM CUSTOMER")

my_data <- dbFetch(my_query, n = -1)

dbClearResult(my_query)

print(my_data)

remove(my_query)
dbDisconnect(db)
```

You can get detailed information on how to work with R and SQL in the official DBI documentation: * DBI Interface: https://cran.r-project.org/web/packages/DBI/index.html * DBI PDF: https://cran.r-project.org/web/packages/DBI/DBI.pdf

6-3 Using SQL with Java/Scala/Kotlin

There are many solutions to make a Java, Scala, or Kotlin application work with a database. The vanilla way we will learn is to use JDBC (Java Database Connection).

Keep in mind there are many solutions and libraries that abstract away SQL operations, which can be good or bad depending on how much control you want to maintain:

- Hibernate - ORM technology that's been around since 2001 and has a mature implementation. However, Hibernate is notorious for its strange loading mechanisms, and can be a hindrance if you want to maintain control of how and when data is loaded from a database.

- jOOQ - A more modern (but commercial) ORM that fluently allows working with databases in a type-safe manner.

- Speedment - Another fast turnaround, fluent API that compiles pure Java code from table schemas to work with databases.

If you are going to go the vanilla JDBC route, it is a good idea to use a connection pool so you can persist and reuse several connections safely in a multithreaded environment. HikariCP is a leading option to achieve this and provides an optimal `DataSource` implementation, which is Java's recommended interface for a database connection pool.

A helpful resource to learning how to work with JDBC is Jenkov's in-depth tutorial:
http://tutorials.jenkov.com/jdbc/index.html

6.3A - Selecting Data with JDBC and HikariCP

To connect to a database using JDBC and HikariCP, you will need the appropriate JDBC drivers for your database platform (e.g. SQLite) as well as Hikari-CP.

```
dependencies {
  compile 'org.xerial:sqlite-jdbc:3.19.3'
  compile 'com.zaxxer:HikariCP:2.6.3'
  compile 'org.slf4j:slf4j-simple:1.7.25'
}
```

Below, we create a simple Java application that creates a Hikari data source with a minimum of 1 connection and a maximum of 5. Then we create a query and loop through it's ResultSet.

```java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class JavaLauncher {

    public static void main(String[] args) {

        try {
            HikariConfig config = new HikariConfig();

config.setJdbcUrl("jdbc:sqlite:/c:/git/oreilly_advanced_sql_for_data/thunderbird_manufacturing.db");
            config.setMinimumIdle(1);
            config.setMaximumPoolSize(5);

            HikariDataSource ds = new HikariDataSource(config);

            Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * from CUSTOMER");

            while (rs.next()) {
                System.out.println(rs.getInt("CUSTOMER_ID") + " " +
rs.getString("CUSTOMER_NAME"));
            }

            //release connection back to pool
```

```
                conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

6.3B Passing parameters

If you need to pass parameters to your SQL query, avoid concatenating the values into the SQL string otherwise you will put your application at risk for SQL injection. Instead, use a PreparedStatement to safely inject the paramters:

```java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class JavaLauncher {

    public static void main(String[] args) {

        try {
            HikariConfig config = new HikariConfig();

config.setJdbcUrl("jdbc:sqlite:/c:/git/oreilly_advanced_sql_for_data/thunderb
ird_manufacturing.db");
            config.setMinimumIdle(1);
            config.setMaximumPoolSize(5);

            HikariDataSource ds = new HikariDataSource(config);

            Connection conn = ds.getConnection();

            // Create a PreparedStatement and populate parameter
            PreparedStatement stmt = conn.prepareStatement("SELECT * FROM
CUSTOMER WHERE CUSTOMER_ID = ?");
            stmt.setInt(1,3);

            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                System.out.println(rs.getInt("CUSTOMER_ID") + " " +
rs.getString("CUSTOMER_NAME"));
            }
```

```java
            //release connection back to pool
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

6-3C Writing Data

You can also use a PreparedStatement to execute updates against the database. The PreparedStatement has more advanced features like batching to write large volumes of data, but here is how to insert a single record.

```java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.PreparedStatement;

public class JavaLauncher {

    public static void main(String[] args) {

        try {
            HikariConfig config = new HikariConfig();

config.setJdbcUrl("jdbc:sqlite:/c:/git/oreilly_advanced_sql_for_data/thunderb
ird_manufacturing.db");
            config.setMinimumIdle(1);
            config.setMaximumPoolSize(5);

            HikariDataSource ds = new HikariDataSource(config);

            Connection conn = ds.getConnection();

            // Create a PreparedStatement and populate parameter
            PreparedStatement stmt =
                    conn.prepareStatement("INSERT INTO PRODUCT
(PRODUCT_NAME,PRODUCT_GROUP,PRICE) VALUES (?,?,?)");

            stmt.setString(1,"Kry Kall");
            stmt.setString(2,"BETA");
            stmt.setBigDecimal(3, BigDecimal.valueOf(35.0));

            stmt.executeUpdate();
```

```java
            //release connection back to pool
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```