# 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\_advanced\_sql\_for\_data

# 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 - 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.

## 2.2C - 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'  
)

## 2.3 - 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\_ID,  
ORDER\_DATE,  
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 - Unions

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

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:

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

## 2.5 - 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 1

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 1

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

## Exercise 2-2

Bring in all records for CUSTOMER\_ORDER, but also bring in the minimum and maximum 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

# 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.

## 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.

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:

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:

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:

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}.

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.

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:

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.

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

To allow 0 or more repetitions, use the \*

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:

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.

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

## 3.4 Wildcards

A dot . represents any character, even whitespaces.

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

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

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:

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:

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

## 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.

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](https://www.techonthenet.com/sql/tables/create_table2.php):

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.

SELECT CUSTOMER\_ORDER.\*,  
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,  
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

## 4.5 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

## 4.6 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:

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.

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. We should only filter the calendar to a date range of interest, like 2017-01-01 and 2017-03-31.

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:

SELECT CALENDAR\_DATE,  
all\_combos.PRODUCT\_ID,  
TOTAL\_QTY  
  
FROM  
(  
 SELECT  
 CALENDAR\_DATE,  
 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

## 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,  
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,  
 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

# Section V - Windowing

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

* Oracle
* Teradata
* PostgreSQL
* SQL Server
* Apache Spark SQL

These platforms notably do not have windowing functions:

* MySQL
* SQLite
* MariaDB

Since SQLite does not support windowing functions, we are going to use [PostgreSQL](https://www.postgresql.org/). While PostgreSQL is free and open-source, there are a few steps in getting it set up. Therefore to save time we are going to use Rextester, a web-based client that can run PostgreSQL queries.

http://rextester.com/l/postgresql\_online\_compiler

In the resources for this class, you should find a "customer\_order.sql" file which can be opened with any text editor. Inside you will see some SQL commands to create and populate a CUSTOMER\_ORDER table and then SELECT all the records from it. Copy/Paste the contents to Rextester and the click the "Run it (F8)" button.

Notice it will create the table and populate it, and the final SELECT query will execute and display the results. Note that the table is not persisted after the operation finishes, so you will need to precede each SELECT exercise with this table creation and population before your SELECT.

## 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 maximum quantity that customer purchased for that PRODUCT\_ID. We can do that with an OVER PARTITION BY combined with the MAX() function.

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.

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

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

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\_customer\_and\_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-day rolling average (3 days before, 3 days after).

Note that PostgreSQL does not support this but MySQL, Teradata, and a few other platforms do.

SELECT CUSTOMER\_ORDER\_ID,  
CUSTOMER\_ID,  
ORDER\_DATE,  
PRODUCT\_ID,  
QUANTITY,  
AVG(QUANTITY) OVER (ORDER BY ORDER\_DATE RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING) AS SIX\_DAY\_WINDOW\_AVG  
  
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.

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:

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))

You can also use these functions to update data.

## 6.2 Using SQL with R

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

setwd('c:\\my\_folder')  
  
library(DBI)  
library(RSQLite)  
  
db <- dbConnect(SQLite(), dbname='thunderbird\_manufacturing.db')  
  
myQuery <- dbSendQuery(db, "SELECT \* FROM CUSTOMER")  
  
my\_data <- dbFetch(myQuery, n = -1)  
  
dbClearResult(myQuery)  
  
print(my\_data)  
  
remove(myQuery)  
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 Java 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](http://hibernate.org/) - 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](https://www.jooq.org/) - A more modern (but commercial) ORM that fluently allows working with databases in a type-safe manner.
* [Speedment](https://www.speedment.com/) - 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](https://github.com/brettwooldridge/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.

import com.zaxxer.hikari.HikariConfig;  
import com.zaxxer.hikari.HikariDataSource;  
  
import java.sql.Connection;  
import java.sql.ResultSet;  
import java.sql.Statement;  
  
public class Launcher {  
  
 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:

import com.zaxxer.hikari.HikariConfig;  
import com.zaxxer.hikari.HikariDataSource;  
  
import java.sql.Connection;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
  
public class Launcher {  
  
 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();  
  
 // 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"));  
 }  
  
 //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.

import com.zaxxer.hikari.HikariConfig;  
import com.zaxxer.hikari.HikariDataSource;  
  
import java.math.BigDecimal;  
import java.sql.Connection;  
import java.sql.PreparedStatement;  
  
public class Launcher {  
  
 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();  
  
 // 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();  
  
 //release connection back to pool  
 conn.close();  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}