

```

1  -- SQL Aggregations Lesson Overview
2  -- In this lesson, we will cover and you will be able to:
3  /*
4      Deal with NULL values
5      Create aggregations in your SQL Queries including:
6          COUNT
7          SUM
8          MIN & MAX
9          AVG
10         GROUP BY
11         DISTINCT
12         HAVING
13     Create DATE functions
14     Implement CASE statements
15 */
16
17 -- NULLs
18 -- NULLs are a datatype that specifies where no data exists in SQL.
19 -- They are often ignored in our aggregation functions like COUNT SUM AVG MIN MAX
20
21 -- NULLs and Aggregation
22
23 -- Notice that NULLs are different than a zero, they are cells where data does not exist.
24 SELECT *
25     FROM accounts
26     WHERE id > 1500 and id < 1600
27     LIMIT 5;
28
29 -- When identifying NULLs in a WHERE clause, we write IS NULL or IS NOT NULL.
30 -- We don't use =, because NULL isn't considered a value in SQL. Rather,
31 -- it is a property of the data.
32
33 -- IS NULL
34 SELECT *
35     FROM accounts
36     WHERE primary_poc IS NULL
37     LIMIT 5;
38
39 -- IS NOT NULL
40 SELECT *
41     FROM accounts
42     WHERE primary_poc IS NOT NULL
43     LIMIT 5;
44
45 -- NULLs - Expert Tip
46
47 -- There are two common ways in which you are likely to encounter NULLs:
48 /*
49     NULLs frequently occur when performing a LEFT or RIGHT JOIN.
50     You saw in the last lesson - when some rows in the left table
51     of a left join are not matched with rows in the right table,
52     those rows will contain some NULL values in the result set.
53 */
54
55 /*
56     NULLs can also occur from simply missing data in our database.
57 */
58
59 -- COUNT the Number of Rows in a Table
60 -- finding the number of rows in each table. Here is an example of finding all the rows.
61 -- ignore NULL VALUES.
62
63 SELECT COUNT(*)
64     FROM accounts;
65
66 SELECT COUNT(accounts.id)
67     FROM accounts;
68
69 SELECT *

```

```

70     FROM orders
71     WHERE occurred_at >= '2016-12-01'
72         AND occurred_at < '2017-01-01';
73
74 SELECT COUNT(*)
75     FROM orders
76     WHERE occurred_at >= '2016-12-01'
77         AND occurred_at < '2017-01-01';
78
79 SELECT COUNT(*) AS order_count
80     FROM orders
81     WHERE occurred_at >= '2016-12-01'
82         AND occurred_at < '2017-01-01';
83
84 -- Notice that COUNT does not consider rows that have NULL values.
85 -- Therefore, this can be useful for quickly identifying which rows have missing data.
86
87 SELECT COUNT (*) AS account_count
88     FROM accounts;
89
90 SELECT COUNT (id) AS account_id_count
91     FROM accounts;
92
93 SELECT COUNT(primary_poc) AS account_primary_poc_count
94     FROM accounts;
95
96 SELECT *
97     FROM accounts
98     WHERE primary_poc IS NULL;
99
100 SELECT count(*)
101     FROM accounts
102     WHERE primary_poc IS NULL;
103
104 -- Unlike COUNT, you can only use SUM on numeric columns. However, SUM will ignore NULL values
105
106 SELECT *
107     FROM orders;
108
109 --summation of quantity of each type of paper
110 SELECT SUM(standard_qty) AS standard,
111        SUM(gloss_qty) AS gloss,
112        SUM.poster_qty) AS poster
113     FROM orders;
114
115 --summation of income of each type of paper
116 SELECT SUM(orders.standard_amt_usd) AS standard_income,
117        SUM(orders.gloss_amt_usd) AS gloss_income,
118        SUM(orders.poster_amt_usd) AS poster_income
119     FROM orders;
120
121 SELECT SUM(standard_qty) AS standard_quantity,
122        SUM(orders.standard_amt_usd) AS standard_income,
123        SUM(gloss_qty) AS gloss_quantity,
124        SUM(orders.gloss_amt_usd) AS gloss_income,
125        SUM(orders.poster_qty) AS poster_quantity,
126        SUM(orders.poster_amt_usd) AS poster_income
127     FROM orders;
128
129 -- Quiz: SUM
130 /*
131 Aggregation Questions:
132     Use the SQL environment below to find the solution for each
133     of the following questions. If you get stuck or want to check
134     your answers, you can find the answers at the top of the next concept.
135 */
136
137 -- Find the total amount of poster_qty paper ordered in the orders table.
138 --code

```

```

139 SELECT sum(orders.poster_qty) AS amount_poster_qty_paper
140     FROM orders;
141
142 -- Find the total amount of standard_qty paper ordered in the orders table.
143 -- code
144 SELECT sum(orders.standard_qty) AS amount_standard_qty_paper
145     FROM orders;
146
147 -- Find the total dollar amount of sales using the total_amt_usd in the orders table.
148 -- code
149 SELECT sum(orders.total_amt_usd) AS total_amt_usd
150     FROM orders;
151
152 -- Find the total amount spent on standard_amt_usd and gloss_amt_usd paper for each order in the orders
table.
153 -- This should give a dollar amount for each order in the table.
154 --code
155 SELECT (orders.standard_amt_usd + orders.gloss_amt_usd) AS total_gloss_and_standard_amt_usd
156     FROM orders;
157
158 -- Find the standard_amt_usd per unit of standard_qty paper.
159 -- Your solution should use both aggregation and a mathematical operator.
160
161 SELECT sum(standard_amt_usd)/sum(standard_qty) AS standard
162     FROM orders;
163
164 -- MIN will return the lowest number.
165 -- MAX does the opposite—it returns the highest number
166 -- MIN and MAX number of orders of each paper type. However, you could run each individually.
167 -- Notice that MIN and MAX are aggregators that again ignore NULL values.
168
169 SELECT MIN(standard_qty) AS standard_min,
170        MIN(gloss_qty) AS gloss_min,
171        MIN.poster_qty AS poster_min,
172        MAX(standard_qty) AS standard_max,
173        MAX(gloss_qty) AS gloss_max,
174        MAX.poster_qty AS poster_max
175     FROM orders;
176
177 -- Similar to other software AVG returns the mean of the data
178 -- that is the sum of all of the values in the column divided by the number of values in a column.
179 -- This aggregate function again ignores the NULL values in both the numerator and the denominator.
180
181 SELECT AVG(standard_qty) AS standard_avg,
182        AVG(gloss_qty) AS gloss_avg,
183        AVG.poster_qty AS poster_avg
184     FROM orders;
185
186 -- Quiz: MIN, MAX, & AVG
187 /*
188 Questions: MIN, MAX, & AVERAGE :
189     Use the SQL environment below to assist with answering the following questions.
190     Whether you get stuck or you just want to double-check your solutions
191     my answers can be found at the top of the next concept.
192 */
193
194 -- When was the earliest order ever placed? You only need to return the date.
195 -- code
196 SELECT min(orders.occurred_at)
197     FROM orders;
198
199 -- Try performing the same query as in question 1 without using an aggregation function.
200 -- code
201 SELECT orders.occurred_at
202     FROM orders
203     ORDER BY orders.occurred_at
204     LIMIT 1;
205
206 -- When did the most recent (latest) web_event occur?

```

```

207 -- code
208 SELECT max(web_events.occurred_at)
209     FROM web_events;
210
211 -- Try to perform the result of the previous query without using an aggregation function.
212 -- code
213 SELECT occurred_at
214     FROM web_events
215     ORDER BY occurred_at DESC
216     LIMIT 1;
217
218 /*
219 Find the mean (AVERAGE) amount spent per order on each paper type,
220 as well as the mean amount of each paper type purchased per order.
221 Your final answer should have 6 values - one for each paper type
222 for the average number of sales, as well as the average amount.
223 */
224 -- code
225 SELECT avg(orders.total_amt_usd/orders.standard_amt_usd),
226        avg(orders.total_amt_usd/orders.gloss_amt_usd),
227        avg(orders.total_amt_usd/orders.poster_amt_usd),
228        avg(orders.total/orders.standard_qty),
229        avg(orders.total/orders.gloss_qty),
230        avg(orders.total/orders.poster_qty)
231     FROM orders;
232
233
234 SELECT AVG(standard_qty) mean_standard, AVG(gloss_qty) mean_gloss,
235        AVG(posters_qty) mean_poster, AVG(standard_amt_usd) mean_standard_usd,
236        AVG(gloss_amt_usd) mean_gloss_usd, AVG(posters_amt_usd) mean_poster_usd
237     FROM orders;
238
239 /*
240 Via the video, you might be interested in
241 how to calculate the MEDIAN. Though this is more advanced
242 than what we have covered so far try finding
243 what is the MEDIAN total_usd spent on all orders?
244 */
245 -- code
246 SELECT *
247     FROM (
248         SELECT total_amt_usd
249             FROM orders
250             ORDER BY total_amt_usd
251             LIMIT 3457) AS Table1
252     ORDER BY total_amt_usd DESC
253     LIMIT 2;
254
255 -- GROUP BY can be used to aggregate data within subsets of the data.
256 -- For example, grouping for different accounts, different regions, or different
257 -- sales representatives.
258
259 -- Any column in the SELECT statement that is not within an aggregator must be in the GROUP BY clause.
260
261 -- The GROUP BY always goes between WHERE and ORDER BY.
262
263 -- ORDER BY works like SORT in spreadsheet software.
264
265 SELECT account_id,
266        SUM(standard_qty) AS standard,
267        SUM(gloss_qty) AS gloss,
268        SUM(posters_qty) AS poster
269     FROM orders;
270
271 SELECT account_id,
272        SUM(standard_qty) AS standard,
273        SUM(gloss_qty) AS gloss,
274        SUM(posters_qty) AS poster
275     FROM orders

```

```

276     GROUP BY account_id
277     ORDER BY account_id;
278
279 -- Quiz: GROUP BY
280 /*
281 GROUP BY Note:
282     Now that you have been introduced to JOINS, GROUP BY, and aggregate
283     functions, the real power of SQL starts to come to life.
284     Try some of the below to put your skills to the test!
285 */
286
287 /*
288 Questions: GROUP BY:
289     Use the SQL environment below to assist with answering the following questions.
290     Whether you get stuck or you just want to double-check your solutions,
291     my answers can be found at the top of the next concept.
292
293     One part that can be difficult to recognize is when it might be
294     easiest to use an aggregate or one of the other SQL functionalities.
295     Try some of the below to see if you can differentiate to find
296     the easiest solution.
297 */
298
299 -- Which account (by name) placed the earliest order?
300 -- Your solution should have the account name and the date of the order.
301 -- code
302 SELECT accounts.name , orders.occurred_at
303     FROM orders
304     JOIN accounts
305         ON accounts.id = orders.account_id
306     ORDER BY orders.occurred_at
307     LIMIT 1;
308
309
310 -- Find the total sales in usd for each account.
311 -- You should include two columns - the total sales for each company's orders in usd and the company name.
312 -- code
313 SELECT accounts.name , orders.total_amt_usd
314     FROM orders
315     JOIN accounts
316         ON accounts.id = orders.account_id
317     ORDER BY accounts.name;
318
319
320 -- Via what channel did the most recent (latest) web_event occur,
321 -- which account was associated with this web_event?
322 -- Your query should return only three values - the date, channel, and account name.
323 -- code
324 SELECT accounts.name , web_events.occurred_at , web_events.channel
325     FROM web_events
326     JOIN accounts
327         ON accounts.id = web_events.account_id
328     GROUP BY accounts.name
329     ORDER BY web_events.occurred_at DESC
330     LIMIT 1;
331
332 -- Find the total number of times each type of channel from the web_events was used.
333 -- Your final table should have two columns ,
334 -- the channel and the number of times the channel was used.
335 -- code
336 SELECT w.channel, COUNT(*)
337     FROM web_events w
338     GROUP BY w.channel
339
340 -- Who was the primary contact associated with the earliest web_event?
341 -- code
342 SELECT w.channel, COUNT(*)
343     FROM web_events w
344     ORDER BY w.channel

```

```

345
346 -- What was the smallest order placed by each account in terms of total usd.
347 -- Provide only two columns - the account name and the total usd.
348 -- Order from smallest dollar amounts to largest.
349 -- code
350 SELECT a.primary_poc
351 FROM web_events w
352 JOIN accounts a
353 ON a.id = w.account_id
354 ORDER BY w.occurred_at
355 LIMIT 1;
356
357 -- Find the number of sales reps in each region.
358 -- Your final table should have two columns ,the region and the number of sales_reps.
359 -- Order from the fewest reps to most reps.
360 -- code
361 SELECT a.name, MIN(total_amt_usd) smallest_order
362 FROM accounts a
363 JOIN orders o
364 ON a.id = o.account_id
365 GROUP BY a.name
366 ORDER BY smallest_order;
367
368 /*
369     You can GROUP BY multiple columns at once, as we showed here.
370     This is often useful to aggregate across a number of different segments.
371 */
372
373 /*
374     The order of columns listed in the ORDER BY clause does make a difference.
375     You are ordering the columns from left to right.
376 */
377 /*
378     GROUP BY - Expert Tips:
379
380     The order of column names in your GROUP BY clause doesn't matter
381     the results will be the same regardless. If we run the same query and reverse
382     the order in the GROUP BY clause, you can see we get the same results.
383
384     As with ORDER BY, you can substitute numbers for column names
385     in the GROUP BY clause. It's generally recommended to do this only
386     when you're grouping many columns, or if something else is causing
387     the text in the GROUP BY clause to be excessively long.
388
389     A reminder here that any column that is not within an aggregation must show up in your GROUP BY
390 statement.
391     If you forget, you will likely get an error. However, in the off chance that your query does work,
392     you might not like the results!
393 */
394 SELECT account_id,
395        channel,
396        COUNT(id) as events
397 FROM web_events
398 GROUP BY account_id, channel
399 ORDER BY account_id, channel;
400
401 SELECT account_id,
402        channel,
403        COUNT(id) as events
404 FROM web_events
405 GROUP BY account_id, channel
406 ORDER BY account_id, channel DESC;
407
408 /*
409     Quiz: GROUP BY Part II
410     Questions: GROUP BY Part II
411     Use the SQL environment below to assist with answering the following questions.
412     Whether you get stuck or you just want to double-check your solutions,

```

```

413     my answers can be found at the top of the next concept.
414 */
415
416
417 -- For each account, determine the average amount of each type of paper they purchased across their orders.
418 -- Your result should have four columns - one for the account name and one for the average quantity
419 --code
420 SELECT accounts.name,
421        AVG(orders.standard_qty) AS avg_stand ,
422        AVG(orders.gloss_qty) AS avg_gloss ,
423        AVG(orders.poster_qty) AS avg_poster
424 FROM accounts
425 JOIN orders
426     ON accounts.id = orders.account_id
427 GROUP BY accounts.name;
428
429 -- For each account, determine the average amount spent per order on each paper type.
430 -- Your result should have four columns - one for the account name and one for the average amount spent
431 -- code
432 SELECT accounts.name,
433        AVG(orders.standard_amt_usd) AS avg_stand_price ,
434        AVG(orders.gloss_amt_usd) AS avg_gloss_price ,
435        AVG(orders.poster_amt_usd) AS avg_poster_price
436 FROM accounts
437 JOIN orders
438     ON accounts.id = orders.account_id
439 GROUP BY accounts.name;
440
441
442 -- Determine the number of times a particular channel was used in the web_events table for each sales rep.
443 -- Your final table should have three columns - the name of the sales rep, the channel, and the number of
444 occurrences.
445 --code
446 SELECT sales_reps.name , count(web_events.channel) AS no_channels ,
447        count(web_events.occurred_at) AS no_occurrences
448 FROM accounts
449 JOIN web_events
450     ON accounts.id = web_events.account_id
451 JOIN sales_reps
452     ON accounts.id = sales_reps.id
453 ORDER BY web_events.occurred_at;
454
455
456
457 -- Determine the number of times a particular channel was used in the web_events table for each region.
458 -- Your final table should have three columns - the region name, the channel, and the number of
459 occurrences.
460 --code
461
462
463
464
465
466
467
468
469
470
471
472 -- DISTINCT is always used in SELECT statements
473 -- it provides the unique rows for all columns written in the SELECT statement.
474 -- Therefore, you only use DISTINCT once in any particular SELECT statement.
475
476 -- which would return the unique (or DISTINCT) rows across all three columns.
477

```

```

478 -- DISTINCT - Expert Tip
479 -- It's worth noting that using DISTINCT, particularly in aggregations,
480 -- can slow your queries down quite a bit.
481
482 SELECT account_id , channel , COUNT(id) as events
483 FROM web_events
484 GROUP BY account_id, channel
485 ORDER BY account_id, channel DESC;
486
487 SELECT account_id , channel
488 FROM web_events
489 GROUP BY account_id, channel
490 ORDER BY account_id;
491
492 SELECT DISTINCT account_id , channel
493 FROM web_events
494 ORDER BY account_id;
495
496 -- Quiz: DISTINCT
497 /*
498 Questions: DISTINCT:
499 Use the SQL environment below to assist with answering the following questions.
500 Whether you get stuck or you just want to double-check your solutions,
501 my answers can be found at the top of the next concept.
502 */
503
504 -- Use DISTINCT to test if there are any accounts associated with more than one region.
505 -- code
506 SELECT DISTINCT
507 FROM accounts
508 JOIN sales_reps
509 ON accounts.sales_rep_id = accounts.id
510 JOIN region
511 ON sales_reps.region_id = region.id
512
513
514 SELECT DISTINCT id, name
515 FROM accounts;
516
517 -- Have any sales reps worked on more than one account?
518 -- code
519 SELECT DISTINCT
520 FROM
521
522 SELECT DISTINCT id, name
523 FROM sales_reps;
524
525 -- HAVING is the "clean" way to filter a query that has been aggregated
526 -- Essentially, any time you want to perform a WHERE on an element of your query that was created by an
527 aggregate, you need to use HAVING instead.
528
529 SELECT account_id , SUM(total_amt_usd) AS sum_total_amt_usd
530 FROM orders
531 GROUP BY account_id
532 HAVING SUM(total_amt_usd) >= 250000;
533
534 /*
535 Questions: HAVING
536 Use the SQL environment below to assist with answering the following questions.
537 Whether you get stuck or you just want to double-check your solutions,
538 my answers can be found at the top of the next concept.
539 */
540
541 -- How many of the sales reps have more than 5 accounts that they manage?
542 -- code
543 SELECT sales_reps.id , sales_reps.name , COUNT(*) AS no_accounts
544 FROM accounts
545 JOIN sales_reps
546 ON accounts.sales_rep_id = sales_reps.id
547 GROUP BY sales_reps.id , sales_reps.name
548 HAVING no_accounts > 5;

```



```

546
547 -- How many accounts have more than 20 orders?
548 -- code
549 SELECT accounts.id , accounts.name , COUNT(*) AS no_orders
550     FROM accounts
551     JOIN orders
552         ON orders.account_id = accounts.id
553     GROUP BY accounts.id , accounts.name
554     HAVING no_orders > 20;
555
556 -- Which account has the most orders?
557 -- code
558 SELECT accounts.id , accounts.name , COUNT(*) AS no_orders
559     FROM accounts
560     JOIN orders
561         ON orders.account_id = accounts.id
562     GROUP BY accounts.id , accounts.name
563     ORDER by no_orders DESC
564     LIMIT 1;
565
566 -- Which accounts spent more than 30,000 usd total across all orders?
567 -- code
568 SELECT accounts.id , accounts.name , orders.total_amt_usd
569     FROM accounts
570     JOIN orders
571         ON orders.account_id = accounts.id
572     GROUP BY accounts.id , accounts.name
573     HAVING orders.total_amt_usd > 30000
574     ORDER BY orders.total_amt_usd DESC;
575
576 -- Which accounts spent less than 1,000 usd total across all orders?
577 -- code
578 SELECT accounts.id , accounts.name , orders.total_amt_usd
579     FROM accounts
580     JOIN orders
581         ON orders.account_id = accounts.id
582     GROUP BY accounts.id , accounts.name
583     HAVING orders.total_amt_usd < 1000
584     ORDER BY orders.total_amt_usd;
585
586 -- Which account has spent the most with us?
587 -- code
588 SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
589     FROM accounts
590     JOIN orders
591         ON orders.account_id = accounts.id
592     GROUP BY accounts.id , accounts.name
593     ORDER BY total_spent DESC
594     LIMIT 1;
595
596 -- Which account has spent the least with us?
597 -- code
598 SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
599     FROM accounts
600     JOIN orders
601         ON orders.account_id = accounts.id
602     GROUP BY accounts.id , accounts.name
603     ORDER BY total_spent
604     LIMIT 1;
605
606 -- Which accounts used facebook as a channel to contact customers more than 6 times?
607 -- code
608 SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
609     FROM accounts
610     JOIN orders
611         ON orders.account_id = accounts.id
612     GROUP BY accounts.id , accounts.name
613     ORDER BY total_spent DESC
614     LIMIT 1;

```

```

615
616 -- Which account used facebook most as a channel?
617 -- code
618 SELECT accounts.id , accounts.name , web_events.channel
619 FROM accounts
620 JOIN web_events
621 ON web_events.account_id = accounts.id
622 WHERE web_events.channel like 'facebook'
623 GROUP BY accounts.id , accounts.name ;
624
625 -- Which channel was most frequently used by most accounts?
626 -- code
627 SELECT accounts.id , accounts.name , web_events.channel , count(*) AS no_channel
628 FROM accounts
629 JOIN web_events
630 ON web_events.account_id = accounts.id
631 WHERE web_events.channel like 'facebook'
632 GROUP BY accounts.id , accounts.name
633 ORDER BY sum_channel DESC
634 LIMIT 10;
635
636 -- DATE_TRUNC
637 -- allows you to truncate your date to a particular part of your date-time column.
638 --
639 --
640 -- DATE_PART
641 -- can be useful for pulling a specific portion of a date, but notice pulling month or day of the week (dow) means that you are no longer keeping the years in order.
642 -- Rather you are grouping for certain components regardless of which year they belonged in.
643
644
645
646
647
648
649 -- this is code without any truncation
650 -- Query 1:
651 SELECT occurred_at , SUM(standard_qty) AS standard_qty_sum
652 FROM orders
653 GROUP BY occurred_at
654 ORDER BY occurred_at;
655
656 SELECT occurred_at , SUM(standard_qty) AS standard_qty_sum ,
657 DATE_TRUNC('day',occurred_at) AS day_trunc
658 -- ,TO_CHAR(day_trunc) AS day_trunc_char
659 FROM orders
660 GROUP BY occurred_at
661 ORDER BY occurred_at;
662
663 SELECT DATE_TRUNC('day',occurred_at) , SUM(standard_qty) AS standard_qty_sum
664 FROM orders
665 GROUP BY DATE_TRUNC('day',occurred_at)
666 ORDER BY DATE_TRUNC('day',occurred_at);
667
668 -- DATE_PART can be useful for pulling a specific portion of a date
669 --
670 -- but notice pulling month or day of the week (dow) ENUMERATION 0--->>6 sun--->>sat
671 --
672 -- means that you are no longer keeping the years in order.
673 --
674 -- Rather you are grouping for certain components regardless of which year they belonged in.
675
676 -- Query 2:
677 SELECT DATE_PART('dow',occurred_at) AS day_of_week , SUM(total) AS total_qty
678 FROM orders
679 GROUP BY day_of_week
680 ORDER BY total_qty;
681
682 /*

```

```

683 Quiz: DATE Functions
684 Questions: Working With DATES
685     Use the SQL environment below to assist with answering the following questions.
686     Whether you get stuck or you just want to double-check your solutions,
687     my answers can be found at the top of the next concept.
688 */
689 --
690 -- Find the sales in terms of total dollars for all orders in each year,
691 -- ordered from greatest to least. Do you notice any trends in the yearly sales totals?
692 -- code
693 SELECT DATE_TRUNC('year', occurred_at) AS yearly_sales ,
694         SUM(orders.total_amt_usd) AS total_dollars
695     FROM orders
696     GROUP BY yearly_sales
697     ORDER BY total_dollars DESC;
698
699 SELECT DATE_PART('year', occurred_at) ord_year, SUM(total_amt_usd) total_spent
700     FROM orders
701     GROUP BY 1
702     ORDER BY 2 DESC;
703
704
705 -- Which month did Parch & Posey have the greatest sales in terms of total dollars?
706 -- Are all months evenly represented by the dataset?
707 -- code
708 SELECT DATE_TRUNC('month', occurred_at) AS month_sales ,
709         SUM(orders.total_amt_usd) AS total_dollars
710     FROM orders
711     GROUP BY month_sales
712     ORDER BY total_dollars DESC;
713
714 SELECT DATE_PART('month', occurred_at) ord_month, SUM(total_amt_usd) total_spent
715     FROM orders
716     GROUP BY 1
717     ORDER BY 2 DESC;
718
719 -- Which year did Parch & Posey have the greatest sales in terms of the total number of orders?
720 -- Are all years evenly represented by the dataset?
721 -- code
722 SELECT DATE_PART('year', occurred_at) ord_year, COUNT(*) total_sales
723     FROM orders
724     GROUP BY 1
725     ORDER BY 2 DESC;
726
727
728 -- Which month did Parch & Posey have the greatest sales in terms of the total number of orders?
729 -- Are all months evenly represented by the dataset?
730
731 -- In which month of which year did Walmart spend the most on gloss paper in terms of dollars?
732 -- code
733 SELECT DATE_TRUNC('month', o.occurred_at) ord_date, SUM(o.gloss_amt_usd) tot_spent
734     FROM orders o
735     JOIN accounts a
736         ON a.id = o.account_id
737     WHERE a.name = 'Walmart'
738     GROUP BY 1
739     ORDER BY 2 DESC
740     LIMIT 1;
741
742 -- =====
743 -- CASE - Expert Tip
744 -- The CASE statement always goes in the SELECT clause.
745 --
746 --
747
748 -- CASE must include the following components:
749 -- WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any of the other
750 -- previous CASE conditions.
751 -- --

```

```

751 -- Query 1:
752     SELECT id , account_id , occurred_at , channel ,
753           CASE WHEN channel = 'facebook' THEN 'yes'
754               END AS is_facebook
755     FROM web_events
756     ORDER BY occurred_at;
757
758 -- Query 2:
759     SELECT id , account_id , occurred_at , channel ,
760           CASE WHEN channel = 'facebook' THEN 'yes'
761               ELSE 'no'
762               END AS is_facebook
763     FROM web_events
764     ORDER BY occurred_at;
765
766 -- Query 3:
767     SELECT id , account_id , occurred_at , channel ,
768           CASE WHEN channel = 'facebook' OR channel = 'direct' THEN 'yes'
769               ELSE 'no'
770               END AS is_facebook
771     FROM web_events
772     ORDER BY occurred_at;
773
774 -- Query 4:
775     SELECT account_id , occurred_at , total ,
776           CASE WHEN total > 500 THEN 'Over 500'
777               WHEN total > 300 THEN '301 - 500'
778               WHEN total > 100 THEN '101 - 300'
779               ELSE '100 or under' END AS total_group
780     FROM orders ;
781
782
783     SELECT
784         CASE WHEN total > 500 THEN 'Over 500'
785             ELSE '500 or under'
786             END AS total_group, COUNT(*) AS order_count
787     FROM orders
788     GROUP BY 1;
789
790
791
792     SELECT COUNT(1) AS orders_over_500_units
793     FROM orders
794     WHERE total > 500;
795
796 /*
797     Quiz: CASE
798     Questions: CASE
799         Use the SQL environment below to assist with answering the following questions.
800         Whether you get stuck or you just want to double-check your solutions,
801         my answers can be found at the top of the next concept.
802 */
803
804 -- Write a query to display for each order, the account ID, the total amount of the order, and the level
of the order - 'Large' or 'Small' - depending on if the order is $3000 or more, or smaller than $3000.
805 -- query to display the number of orders in each of three categories, based on the total number of items
in each order. The three categories are: 'At Least 2000', 'Between 1000 and 2000' and 'Less than 1000'.
806 -- code
807     SELECT
808         CASE WHEN total >= 2000 THEN 'At Least 2000'
809             WHEN total >= 1000 AND total < 2000 THEN 'Between 1000 and 2000'
810             ELSE 'Less than 1000' END AS order_category,
811         COUNT(*) AS order_count
812     FROM orders
813     GROUP BY 1;
814
815 -- We would like to understand 3 different levels of customers based on the amount associated with their
purchases.
816 -- The top-level includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd.

```

```

817 -- The second level is between 200,000 and 100,000 usd. The lowest level is anyone under 100,000 usd.
818 -- Provide a table that includes the level associated with each account. You should provide the account
      name, the total sales of all orders for the customer, and the level.
819 -- Order with the top spending customers listed first.
820 -- code
821 SELECT a.name, SUM(total_amt_usd) total_spent,
822        CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
823              WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
824              ELSE 'low' END AS customer_level
825 FROM orders o
826 JOIN accounts a
827 ON o.account_id = a.id
828 GROUP BY a.name
829 ORDER BY 2 DESC;
830
831
832 -- We would now like to perform a similar calculation to the first,
833 -- but we want to obtain the total amount spent by customers only in 2016 and 2017.
834 -- Keep the same levels as in the previous question. Order with the top spending customers listed first.
835 -- code
836 SELECT a.name, SUM(total_amt_usd) total_spent,
837        CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
838              WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
839              ELSE 'low' END AS customer_level
840 FROM orders o
841 JOIN accounts a
842 ON o.account_id = a.id
843 WHERE occurred_at > '2015-12-31'
844 GROUP BY 1
845 ORDER BY 2 DESC;
846 -- We would like to identify top-performing sales reps, which are sales reps associated with more than
      200 orders.
847 -- Create a table with the sales rep name, the total number of orders, and a column with top or not
      depending on
848 -- if they have more than 200 orders. Place the top salespeople first in your final table.
849 -- code
850
851
852
853
854
855
856
857
858
859 -- The previous didn't account for the middle, nor the dollar amount associated with the sales.
860 -- Management decides they want to see these characteristics represented as well.
861 -- We would like to identify top-performing sales reps,
862 -- which are sales reps associated with more than 200 orders or more than 750000 in total sales.
863 -- The middle group has any rep with more than 150 orders or 500000 in sales.
864 -- Create a table with the sales rep name, the total number of orders, total sales across all orders, and
      a column with top, middle, or low depending on these criteria.
865 -- Place the top salespeople based on the dollar amount of sales first in your final table. You might see
      a few upset salespeople by this criteria!
866 -- code

```