Week 5: Data Cleaning and Window Functions in SQL

Parch and Posey Database

Contents

Rollups (Northwind Database)

Often when we are aggregating data, we are aggregating across multiple dimensions. For example, if I want to see the total order value for each customer for each product category, I would want to write something like this:

```sql
SELECT ca.categoryname, c.companyname, SUM(od.unitprice * od.quantity) AS total_value
FROM products p
JOIN order_details od ON od.productid = p.productid
JOIN orders o ON o.orderid = od.orderid
JOIN customers c ON c.customerid = o.customerid
JOIN categories ca ON ca.categoryid = p.categoryid
GROUP BY 1,2
ORDER BY 3 DESC
LIMIT 5;
```

categoryname	companyname	total_value
Beverages	QUICK-Stop	38272.80
Meat/Poultry	Save-a-lot Markets	30793.74
Dairy Products	Ernst Handel	26629.30
Meat/Poultry	Hungry Owl All-Night Grocers	24040.90
Dairy Products	Save-a-lot Markets	23774.10

This gets us the correct results. But we often in the same query want to also answer - what is the total order value: 1. For each customer? 2. For each product category?

We can write our query first to get the total order value per category:

```
CREATE OR REPLACE VIEW totals_per_category AS
SELECT ca.categoryname,
       NULL                              AS customername,
       SUM(od.unitprice * od.quantity) AS total_value
FROM products p
         JOIN order_details od ON od.productid = p.productid
         JOIN orders o ON o.orderid = od.orderid
         JOIN categories ca ON ca.categoryid = p.categoryid
GROUP BY 1
ORDER BY 2 DESC
LIMIT 5;
```

For now, ignore the fact that we are just making all values for the `customername` column `NULL`.

Then, we can make more or less the same query to get the total for each account:

```
CREATE OR REPLACE VIEW totals_per_customer AS
SELECT NULL                              AS categoryname,
       c.companyname                     AS customername,
       SUM(od.unitprice * od.quantity) AS total_value
FROM order_details od
         JOIN orders o ON o.orderid = od.orderid
         JOIN customers c ON c.customerid = o.customerid
GROUP BY 2
ORDER BY 2 DESC
LIMIT 5;
```

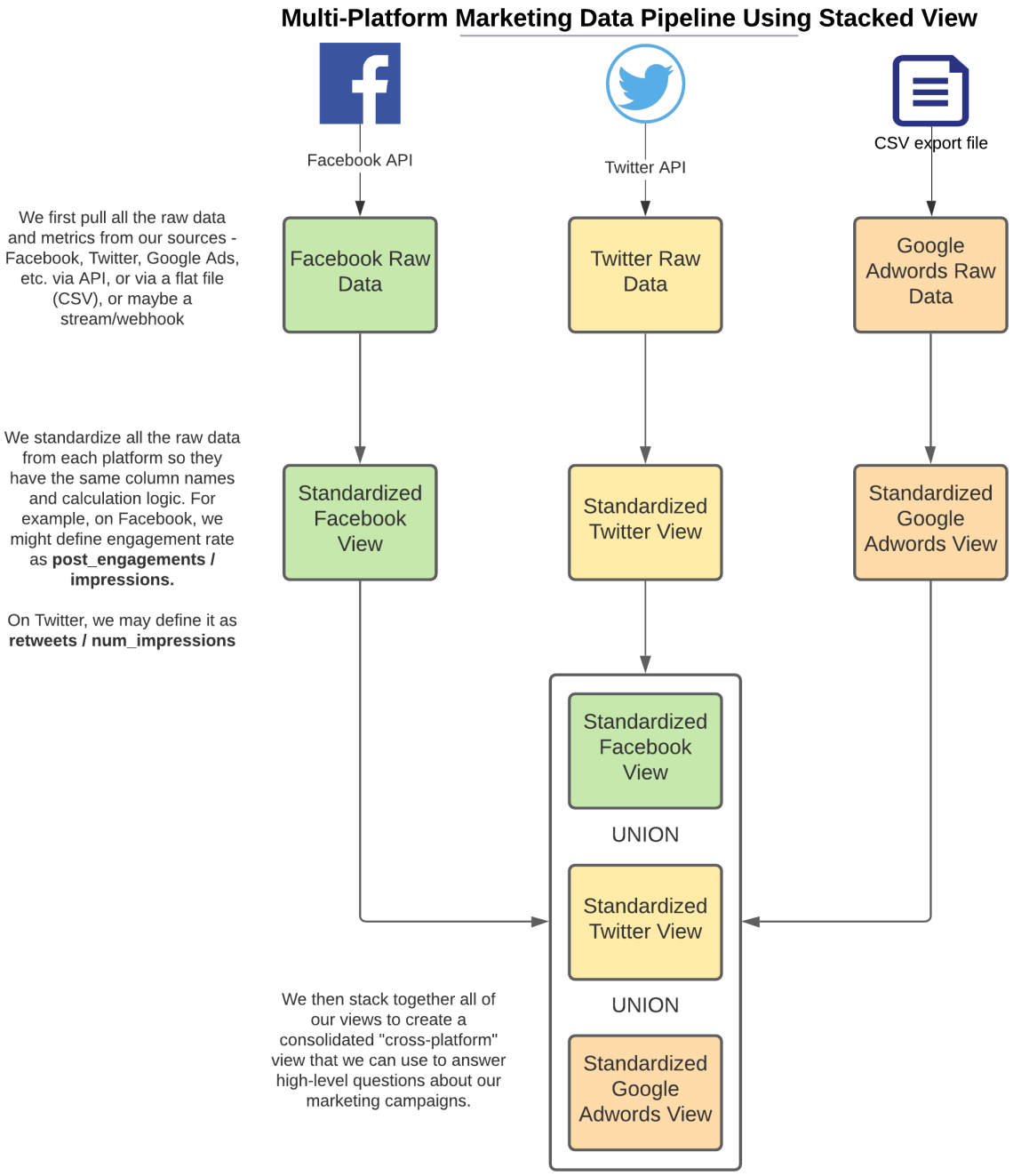
Again, ignore the fact that the `categoryname` is completely null.

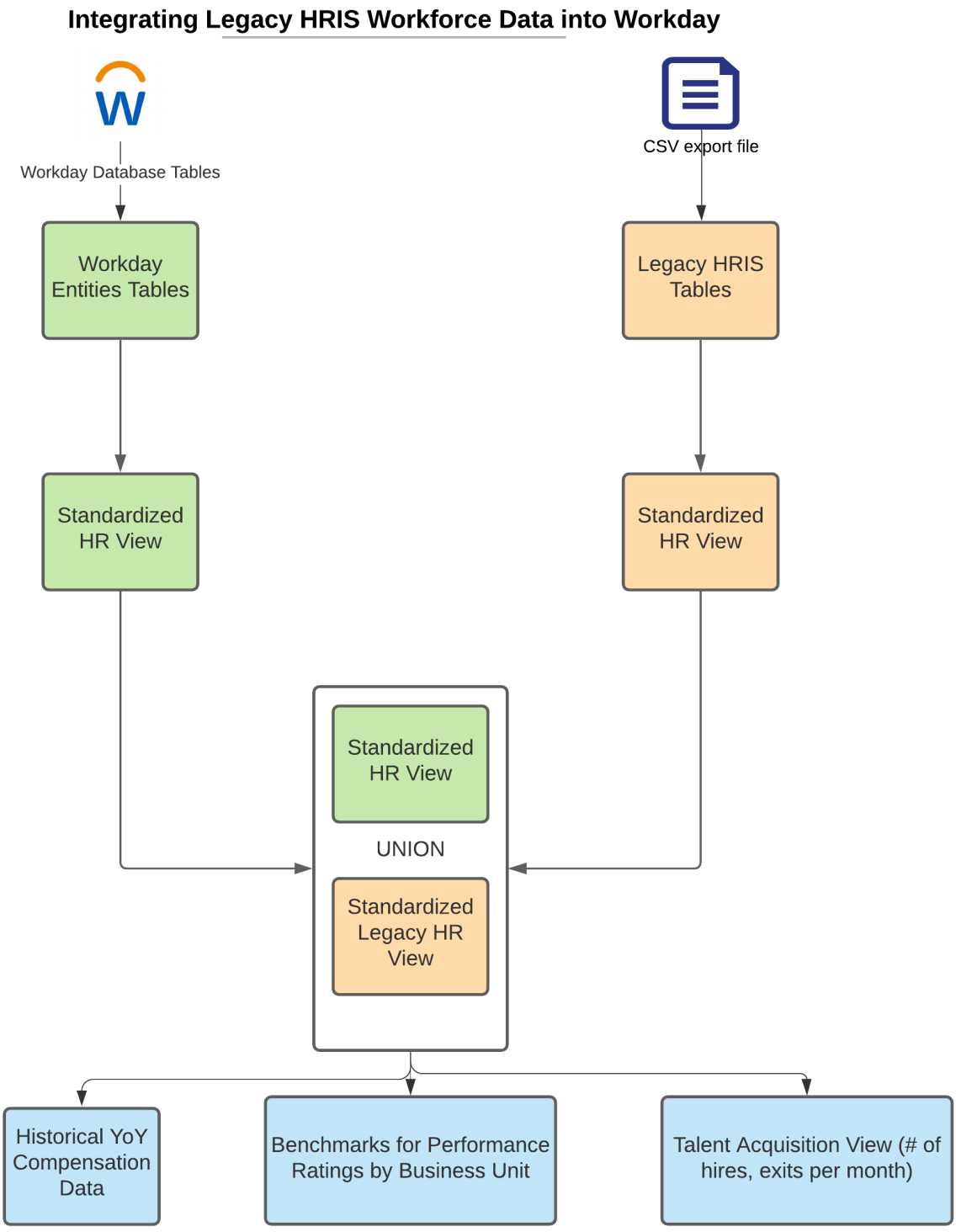
We can combine all of these results into one result set by using `UNION`:

```
SELECT * FROM totals_per_customer

    UNION -- use UNION to stack two result sets together

SELECT * FROM totals_per_category
```

categoryname	customername	total_value
Produce	NA	105268.60
NA	Wolski Zajazd	3531.95
Grains/Cereals	NA	100726.80
NA	Wartian Herkku	16617.10
NA	Wilman Kala	3161.35
Beverages	NA	286526.95
Condiments	NA	113694.75
Meat/Poultry	NA	178188.80
NA	Wellington Importadora	6480.70
NA	White Clover Markets	29073.45

Note how we used the `UNION` keyword to stack together the results of two different views. This is a very common pattern that conforms to the **ELT (Extract-Load-Transform)** process of data pipelines.


Multi-Platform Marketing Data Pipeline Using Stacked View
Facebook API
Twitter API
CSV export file
We first pull all the raw data and metrics from our sources - Facebook, Twitter, Google Ads, etc. via API, or via a flat file (CSV), or maybe a stream/webhook
Facebook Raw Data
Twitter Raw Data
Google Adwords Raw Data
We standardize all the raw data from each platform so they have the same column names and calculation logic. For example, on Facebook, we might define engagement rate as post_engagements / impressions.
On Twitter, we may define it as retweets / num_impressions
Standardized Facebook View
Standardized Twitter View
Standardized Google Adwords View
Standardized Facebook View
UNION
Standardized Twitter View
UNION
Standardized Google Adwords View
We then stack together all of our views to create a consolidated "cross-platform" view that we can use to answer high-level questions about our marketing campaigns.



ROLLUP

However, there's an easier way to do "rollups" of aggregations:

```
SELECT ca.categoryname, c.companyname, SUM(od.unitprice * od.quantity) AS total_value
FROM products p
```

```
JOIN order_details od ON od.productid = p.productid
JOIN orders o ON o.orderid = od.orderid
JOIN customers c ON c.customerid = o.customerid
JOIN categories ca ON ca.categoryid = p.categoryid
GROUP BY ROLLUP (categoryname, companyname)
LIMIT 10;
```

There will be some rows where the `companyname` is `NULL` (the total per category), and some rows where the `categoryname` is `NULL` (the total per account). There will be one row where both `companyname` and `categoryname` are `NULL` - this corresponds the global total.

categoryname	companyname	total_value
NA	NA	1354458.59
Meat/Poultry	LILA-Supermercado	1301.00
Beverages	Reggiani Caseifici	225.00
Condiments	QUICK-Stop	9691.70
Confections	La corne d'abondance	1000.15
Meat/Poultry	Bottom-Dollar Markets	2082.00
Produce	Reggiani Caseifici	1636.80
Dairy Products	HILARION-Abastos	7414.40
Condiments	Island Trading	1655.00
Meat/Poultry	Familia Arquibaldo	1187.50

Data Cleaning

LEFT and RIGHT

You can use the `LEFT` and `RIGHT` functions to extract the last `X` number of characters from a text field. For example, `LEFT(some_column, 2)` will return the first two characters from the `some_column` field.

1. In the `accounts` table, there is a column holding the website for each company. The last three digits specify what type of web address they are using. Pull these extensions and provide how many of each website type exist in the accounts table.

```
SELECT DISTINCT RIGHT(website, 3) AS domain, COUNT(*) num_companies
FROM accounts
GROUP BY 1
ORDER BY 2 DESC
```

domain	num_companies
com	349
net	1
org	1

2. There is much debate about how much the name (or even the first letter of a company name) matters. Use the accounts table to pull the first letter of each company name to see the distribution of company names that begin with each letter (or number).

```
SELECT LEFT(UPPER(name), 1) AS letter, count(*)
FROM accounts
GROUP BY 1
ORDER BY 1
```

letter	count
3	1
A	37
B	16
C	37
D	17
E	16
F	12
G	14
H	15
I	7
J	7
K	7
L	16
M	22
N	15
O	7
P	27
Q	1
R	8
S	17
T	17
U	13
V	7
W	12
X	2
Y	1

3. Use the `accounts` table and a `CASE` statement to create two groups: one group of company names that start with a number and a second group of those company names that start with a letter. What proportion of company names start with a letter?

```
SELECT TYPE, count(*)
FROM (
         SELECT LEFT(UPPER(name), 1) AS letter,
                CASE
                    WHEN LEFT(UPPER(name), 1) IN
                         ('0', '1', '2', '3', '4',
                         '5', '6', '7', '8', '9') THEN 'numeric'
                    ELSE 'character'
                    END              AS TYPE
         FROM accounts
         ORDER BY 1) AS tbl1
GROUP BY 1
```

type	count
character	350
numeric	1

POSITION & STRPOS

4. Use the `accounts` table to create first and last name columns that hold the first and last names for the `primary_poc`.

```
SELECT primary_poc,
       LEFT(primary_poc, position(' ' in primary_poc) -1 ) AS firstname,
       RIGHT(primary_poc, length(primary_poc) - position(' ' in primary_poc) ) AS lastname
FROM accounts
LIMIT 5
```

primary_poc	firstname	lastname
Tamara Tuma	Tamara	Tuma
Sung Shields	Sung	Shields
Jodee Lupo	Jodee	Lupo
Serafina Banda	Serafina	Banda
Angeles Crusoe	Angeles	Crusoe

CONCAT

5. Each company in the `accounts` table wants to create an email address for each `primary_poc`. The email address should be the first name of the `primary_poc` last name `primary_poc` @ company name .com.

```
WITH t1 AS (
  SELECT primary_poc,
    LEFT(primary_poc, position(' ' in primary_poc) -1 ) AS firstname,
    RIGHT(primary_poc, length(primary_poc) - position(' ' IN primary_poc)) AS lastname,
    name
  FROM accounts
    )
SELECT LOWER(CONCAT(firstname, '.', lastname, '@', name, 'com'))
FROM t1
LIMIT 5
```

lower
tamara.tuma@walmartcom
sung.shields@exxon mobilcom
jodee.lupo@applecom
serafina.banda@berkshire hathawaycom
angeles.crusoe@mckessoncom

6. You may have noticed that in the previous solution some of the company names include spaces, which will certainly not work in an email address. See if you can create an email address that will work by removing all of the spaces in the account name, but otherwise your solution should be just AS in the previous question. (Hint: look up `REPLACE`)

```
with t1 AS (
  SELECT primary_poc,
    LEFT(primary_poc, position(' ' in primary_poc) -1 ) AS firstname,
    RIGHT(primary_poc, length(primary_poc) - position(' ' in primary_poc)) AS lastname,
    REPLACE(name, ' ', '') AS company
    FROM accounts
    )
SELECT LOWER(CONCAT(firstname, '.', lastname, '@', company, 'com'))
FROM t1
LIMIT 5
```

lower
tamara.tuma@walmartcom
sung.shields@exxonmobilcom
jodee.lupo@applecom
serafina.banda@berkshirehathawaycom
angeles.crusoe@mckessoncom

7. We would also like to create an initial password, which they will change after their first log in. The first password will be the first letter of the `primary_poc`'s first name (lowercase), then the last letter of their first name (lowercase), the first letter of their last name (lowercase), the last letter of their last name (lowercase), the number of letters in their first name, the number of letters in their last name, and then the name of the company they are working with, all capitalized with no spaces.

```
WITH t1 AS (
  SELECT primary_poc,
    LEFT(primary_poc, position(' ' in primary_poc) -1 ) AS firstname,
    RIGHT(primary_poc, length(primary_poc) - position(' ' in primary_poc)) AS lastname,
    REPLACE(name, ' ', '') AS company
    FROM accounts
    )
SELECT  LOWER(CONCAT(firstname, '.', lastname, '@', company, 'com')) AS email,
        CONCAT(
          LOWER(LEFT(firstname, 1)),
          LOWER(RIGHT(firstname, 1)),
          LOWER(LEFT(lastname,1)),
          LOWER(RIGHT(lastname,1)),
          length(LEFT(primary_poc, position(' ' in primary_poc) -1 )),
          UPPER(REPLACE(company, ' ', ''))
        ) AS password
FROM t1
LIMIT 5
```

email	password
tamara.tuma@walmartcom	tata6WALMART
sung.shields@exxonmobilcom	sgss4EXXONMOBIL
jodee.lupo@applecom	jelo5APPLE
serafina.banda@berkshirehathawaycom	saba8BERKSHIREHATHAWAY
angeles.crusoe@mckessoncom	asce7MCKESSON

More Advanced Join Logic (CLUB Database)

The following exercises will use the `club` database, which has been added to both the primary AWS database and the secondary Hong Kong database.

So far, we've been writing joins using the familiar syntax of `JOIN ... ON ... = ....` However, there are some times when this can become quite tedious, especially if it should be "obvious" what the join key is on. For example, let's try to get the count of the number of bookings by each member in the club:

```
SELECT firstname || ' ' || surname AS member_name, COUNT(*) bookings
FROM bookings b
JOIN members m ON m.memid = b.memid
GROUP BY 1
LIMIT 5;
```

The results are below:

member_name	bookings
Timothy Baker	166
Ramnaresh Sarwin	70
John Hunt	15
David Farrell	34
Hyacinth Tupperware	16

It should be "obvious" that `memid` in both tables reference the same entity - members.

`USING`

Whenever the joining columns in the two tables you wish to join have the same name and you want to avoid having to type `ON ... b.memid = m.memid`, you can use `USING`:

```
SELECT firstname || ' ' || surname AS member_name, COUNT(*) bookings
FROM bookings b JOIN members m
USING (memid)
GROUP BY 1
LIMIT 5;
```

The most are specifying that we want to use `memid` as the join column. Note that we *must* put the join column(s) in parenthesis to indicate we are working with a tuple data structure - this is because it is possible for us to do a join on more than one column.

8. **YOUR TURN** Using the `USING`, show the top 5 facilities in terms of number of bookings.

name	num_bookings
Pool Table	837
Massage Room 1	629
Snooker Table	444
Squash Court	440
Tennis Court 1	408

Natural Joins

It is possible to write even less code to perform joins.

When two tables are linked via matching column names like `memid`, then you can use a `NATURAL JOIN`:

```
SELECT firstname || ' ' || surname AS member_name, COUNT(*) bookings
FROM bookings b NATURAL JOIN members m
GROUP BY 1
LIMIT 5;
```

We'll get the exact same results as using the first query, with less typing.

Let's try another example. We'll need to find the top five facilities in terms of distinct number of members booked all time. We can write the following query to join from `members` to `bookings` to `facilities`:

```
SELECT f.name, COUNT(DISTINCT m.memid) AS members_booked
FROM bookings b
NATURAL JOIN members m
NATURAL JOIN facilities f
GROUP BY 1
```

```
ORDER BY 2 DESC
LIMIT 5;
```

name	members_booked
Pool Table	28
Table Tennis	26
Massage Room 1	25
Badminton Court	25
Squash Court	25

When we use `NATURAL JOIN`, do not use an `ON` or `USING` clause.

9. **YOUR TURN** Using a `NATURAL JOIN`, show the top 3 customers in terms of revenue generated for the club. Revenue is defined as the sum of `membercost`.

memid	firstname	surname	total_revenue
0	GUEST	GUEST	12142.5
3	Tim	Rownam	2930.0
8	Tim	Boothe	1582.0

SELF JOINS (CLUB Database)

10. Create a list of all members who have recommended another member? Ensure that there are no duplicates in the list, and that results are ordered by (`surname`, `firstname`).

```
SELECT DISTINCT recs.firstname, recs.surname
FROM members mems
JOIN members recs
ON recs.memid = mems.recommendedby
```

firstname	surname
Darren	Smith
Timothy	Baker
Florence	Bader
Ponder	Stibbons
Tim	Rownam
Tracy	Smith
Burton	Tracy
Jemima	Farrell
Matthew	Genting
David	Jones
Janice	Joplette
Gerald	Butters
Millicent	Purview

11. **YOUR TURN** Create a list of all members, including the individual who recommended them (if any)? Ensure that results are ordered by (`surname`, `firstname`).

memfname	memsname	recfname	recsname
Florence	Bader	Ponder	Stibbons
Anne	Baker	Ponder	Stibbons
Timothy	Baker	Jemima	Farrell
Tim	Boothe	Tim	Rownam
Gerald	Butters	Darren	Smith
Joan	Coplin	Timothy	Baker
Erica	Crumpet	Tracy	Smith
Nancy	Dare	Janice	Joplette
Matthew	Genting	Gerald	Butters
John	Hunt	Millicent	Purview
David	Jones	Janice	Joplette
Douglas	Jones	David	Jones
Janice	Joplette	Darren	Smith
Anna	Mackenzie	Darren	Smith
Charles	Owen	Darren	Smith
David	Pinker	Jemima	Farrell
Millicent	Purview	Tracy	Smith
Henrietta	Rumney	Matthew	Genting
Ramnaresh	Sarwin	Florence	Bader
Jack	Smith	Darren	Smith
Ponder	Stibbons	Burton	Tracy
Henry	Worthington-Smyth	Tracy	Smith

12. Create an `employee` table in your personal schema for `club`. Make sure that the employee table contains two constraints:

1. A primary key called `employee_id` to uniquely identify each individual employee.
2. A foreign key called `manager_id` which references the primary of itself.

Using the following DDL (Database Definition Language) to do so:

```
CREATE TABLE employee (
 employee_id INT PRIMARY KEY,
 first_name VARCHAR (255) NOT NULL,
 last_name VARCHAR (255) NOT NULL,
 manager_id INT,
 FOREIGN KEY (manager_id)
 REFERENCES employee (employee_id)
);
INSERT INTO employee (
 employee_id,
 first_name,
 last_name,
 manager_id
)
VALUES
 (1, 'Windy', 'Hays', NULL),
 (2, 'Ava', 'Christensen', 1),
 (3, 'Hassan', 'Conner', 1),
 (4, 'Anna', 'Reeves', 2),
 (5, 'Sau', 'Norman', 2),
 (6, 'Kelsie', 'Hays', 3),
 (7, 'Tory', 'Goff', 3),
```

```
 (8, 'Salley', 'Lester', 3);
```

The `FOREIGN KEY (manager_id)` specifies that the `manager_id` column should be a foreign key column. The `REFERENCES employee (employee_id)` indicates that this `manager_id` column should reference the primary key `employee_id` in the `employee` table.

13. Find out who reports to who in the `employee` table you created.

```
SELECT
    e.first_name || ' ' || e.last_name employee,
    m .first_name || ' ' || m .last_name manager
FROM
    employee e
JOIN employee m ON m .employee_id = e.manager_id
ORDER BY
    manager;
```

employee	manager
Sau Norman	Ava Christensen
Anna Reeves	Ava Christensen
Salley Lester	Hassan Conner
Kelsie Hays	Hassan Conner
Tory Goff	Hassan Conner
Ava Christensen	Windy Hays
Hassan Conner	Windy Hays

14. What would you do in order to show the top manager who is not reporting to anyone?

```
SELECT
    e.first_name || ' ' || e.last_name employee,
    m .first_name || ' ' || m .last_name manager
FROM
    employee e
LEFT JOIN employee m ON m .employee_id = e.manager_id
ORDER BY
    manager
```

employee	manager
Anna Reeves	Ava Christensen
Sau Norman	Ava Christensen
Salley Lester	Hassan Conner
Kelsie Hays	Hassan Conner
Tory Goff	Hassan Conner
Hassan Conner	Windy Hays
Ava Christensen	Windy Hays
Windy Hays	NA

Window Functions

PostgreSQL's documentation does an excellent job of introducing the concept of Window Functions:

"A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike

regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result."

15. Print the total number `standard_amt_usd` across all orders next to each order's value.

```
SELECT standard_amt_usd,
       SUM(standard_amt_usd) OVER () AS all_time_total
FROM orders
LIMIT 10
```

standard_amt_usd	all_time_total
613.77	9672347
948.10	9672347
424.15	9672347
718.56	9672347
538.92	9672347
513.97	9672347
503.99	9672347
474.05	9672347
454.09	9672347
469.06	9672347

Notice that the column `all_time_total` is always the same value.

16. Create a running total of `standard_amt_usd` (in the orders table) over order time.

```
SELECT standard_amt_usd,
       SUM(standard_amt_usd) OVER (ORDER BY occurred_at) AS running_total
FROM orders
limit 10
```

standard_amt_usd	running_total
0.00	0.00
2445.10	2445.10
2634.72	5079.82
0.00	5079.82
2455.08	7534.90
2504.98	10039.88
264.47	10304.35
1536.92	11841.27
374.25	12215.52
1402.19	13617.71

17. Create a running total of `standard_amt_usd` (in the orders table) over order time for each month.

```
SELECT standard_amt_usd, date_trunc('month', occurred_at),
       SUM(standard_amt_usd) OVER (PARTITION BY date_trunc('month', occurred_at)
                                   ORDER BY occurred_at) AS running_total
FROM orders
limit 10
```

18. **YOUR TURN** Create a running total of `standard_amt_usd` (in the orders table) over order time for each year.

standard_amt_usd	running_total
0.00	0.00
2445.10	2445.10
2634.72	5079.82
0.00	5079.82
2455.08	7534.90
2504.98	10039.88
264.47	10304.35
1536.92	11841.27
374.25	12215.52
1402.19	13617.71

19. Return all columns from the accounts table, and include a field that has the total number of orders each account has placed.

```
SELECT DISTINCT COUNT(*) OVER (PARTITION BY a.id) AS total_orders_placed, a.*
FROM accounts a
JOIN orders o ON o.account_id = a.id
LIMIT 5
```

total_orders_placed	id	name	website	lat	long	prima
27	1851	Gilead Sciences	www.gilead.com	42.35445	-71.05788	Nickie
12	2151	Time Warner Cable	www.twc.com	41.87656	-87.63267	Jeanne
4	4331	Tenneco	www.tenneco.com	36.16963	-115.13576	Angele
5	4481	Newmont Mining	www.newmont.com	45.49412	-122.66946	Khadij
6	1991	Northwestern Mutual	www.northwesternmutual.com	42.36339	-71.05213	Cori E

20. **YOUR TURN** Show the region table, including the number of sales reps assigned to that region.

id	name	sales_reps
4	West	10
2	Midwest	9
3	Southeast	10
1	Northeast	21

Notice that you cannot use a window function alongside a `GROUP BY`. For example, may seem like a good way to get the account total alongside the global total for orders, but it will yield an error:

```
SELECT a.name,
    SUM(standard_amt_usd) OVER () AS global_total,
    SUM(standard_amt_usd) AS account_total
FROM orders o
JOIN accounts a ON a.id = o.account_id
GROUP BY 1,2;
```

The error is pretty clear:

```
ERROR:  window functions are not allowed in GROUP BY
LINE 2:  SUM(standard_amt_usd) OVER () AS global_total,
```

Instead, what you'd write to get the global total value for all orders is

```
WITH global_total AS (SELECT SUM(standard_amt_usd) FROM orders)
SELECT a.name,
       (SELECT * FROM global_total)                          AS global_total,
```

```
       SUM(standard_amt_usd)                                    AS account_total,
       SUM(standard_amt_usd) / (SELECT * FROM global_total) AS percent_of_revenue
FROM orders o
         JOIN accounts a ON a.id = o.account_id
GROUP BY 1, 2
ORDER BY percent_of_revenue DESC
LIMIT 5;
```

This now allows you to calculate the percentage of revenue coming from each account:

id	name	sales_reps
4	West	10
2	Midwest	9
3	Southeast	10
1	Northeast	21

21. **YOUR TURN** - Create a running total of `standard_qty` (in the orders table) over order time for each year.

standard_qty	month	running_total
0	2013-01-01	0
490	2013-01-01	490
528	2013-01-01	1018
0	2013-01-01	1018
492	2013-01-01	1510
502	2013-01-01	2012
53	2013-01-01	2065
308	2013-01-01	2373
75	2013-01-01	2448
281	2013-01-01	2729

22. **YOUR TURN** - Create a result set that shows order information - `account_id`, `total_amt_usd`, and also the order number (ie. the first order ever for an account is 1, the second is 2). However, only consider orders where the total `total_amt_usd` is greater than 10000.

account_id	total_amt_usd	occurred_at	big_order_number
1081	12352.77	2015-09-07 15:39:04	1
1111	27902.70	2015-11-15 17:47:46	1
1121	18618.65	2015-10-24 15:55:41	1
1141	26055.46	2016-12-21 15:52:58	1
1161	24885.98	2016-06-04 08:58:10	1

23. For many of our accounts, reaching 100,000 in total revenue (as measured by `total_amt_usd`) is a big occasion. List for each account that has reached 100,000 in total revenue the order that pushed their total above 100,000.

```
WITH running_totals AS (
    -- CTE #1
    SELECT id,
           account_id,
           occurred_at,
           SUM(total_amt_usd) OVER (PARTITION BY account_id
                ORDER BY occurred_at) AS running_total
```

```
    FROM orders),
    -- CTE #2
     orders_over_100000 AS (
         SELECT *, ROW_NUMBER() OVER (PARTITION BY account_id) AS order_number_over_100k
         FROM running_totals
         WHERE running_total > 100000)

SELECT id, account_id, occurred_at, running_total
FROM orders_over_100000
WHERE order_number_over_100k = 1
LIMIT 10;
```

id	account_id	occurred_at	running_total
12	1001	2016-08-28 07:13:39	101179.4
49	1081	2016-02-01 20:00:37	100854.2
4380	1121	2016-10-10 10:40:21	100994.6
176	1181	2016-04-24 16:47:51	105261.7
4477	1261	2015-11-27 01:46:04	106093.3
314	1281	2016-02-13 23:55:39	102202.6
339	1291	2016-06-20 02:46:58	100511.7
4532	1301	2015-01-24 06:07:44	104419.8
4563	1341	2016-11-24 21:17:12	101570.9
438	1371	2016-11-01 12:45:09	100555.9

ROW NUMBER and RANK

`ROW_NUMBER()` does just what it sounds like— displays the number of a given row. It starts are 1 and numbers the rows according to the ORDER BY part of the window statement. `ROW_NUMBER()` does not require you to specify a variable within the parentheses. Using the `PARTITION BY` clause will allow you to begin counting 1 again in each partition.

24. Create a new column called `row_number` that gives you the row numbers of orders (ordered by `occurred_at`) for each month (i.e. the count of rows starts with every month)

```
SELECT id,
       account_id,
       date_trunc('month', occurred_at)::DATE AS month,
       row_number() OVER (
           PARTITION BY date_trunc('month', occurred_at)::DATE
           ORDER BY occurred_at)              AS row_num
FROM orders
LIMIT 10;
```

id	account_id	month	row_num
5786	2861	2013-12-01	1
2415	2861	2013-12-01	2
4108	4311	2013-12-01	3
4489	1281	2013-12-01	4
287	1281	2013-12-01	5
1946	2481	2013-12-01	6
6197	3431	2013-12-01	7
3122	3431	2013-12-01	8
6078	3251	2013-12-01	9
2932	3251	2013-12-01	10

What about if you wanted to find the order number for each account for each month? For example, what if you wanted to find the 1st order for each account in each month?

```
SELECT id,
       account_id,
       date_trunc('month', occurred_at)::DATE AS month,
       row_number() OVER (
           PARTITION BY
               date_trunc('month', occurred_at)::DATE,
               account_id
           ORDER BY occurred_at)              AS row_num
FROM orders
LIMIT 10;
```

Notice that all we did was add `account_id` to the group in `PARTITION BY`.

id	account_id	month	row_num
147	1181	2013-12-01	1
243	1251	2013-12-01	1
4457	1251	2013-12-01	2
4489	1281	2013-12-01	1
287	1281	2013-12-01	2
346	1301	2013-12-01	1
4523	1301	2013-12-01	2
4587	1401	2013-12-01	1
446	1401	2013-12-01	2
484	1411	2013-12-01	1

Rank

The function `RANK` works very similarly to `ROW_NUMBER()`. However, "equal" rows are ranked the same. This can be useful when there are ties in your ordering and you want to make sure that these values are not arbitrarily differentiated.

25. Select the `id`, `account_id`, and total variable from the orders table, then create a new column called `rank` that gives you the ranking of orders (ordered by month) for each account.

```
SELECT id,
       account_id,
       date_trunc('month', occurred_at)              AS month,
       rank() OVER (
           PARTITION BY account_id
```

```
          ORDER BY date_trunc('month', occurred_at)) AS row_num
FROM orders
LIMIT 10
```

id	account_id	month	row_num
1	1001	2015-10-01	1
4307	1001	2015-11-01	2
2	1001	2015-11-01	2
3	1001	2015-12-01	4
4308	1001	2015-12-01	4
4309	1001	2016-01-01	6
4	1001	2016-01-01	6
4310	1001	2016-02-01	8
5	1001	2016-02-01	8
6	1001	2016-03-01	10

26. **YOUR TURN** - Select the `id`, `account_id`, and total variable from the `orders` table, then create a column called `total_rank` that ranks this total amount of paper ordered (from highest to lowest) for each account.

id	account_id	total	total_rank
4308	1001	1410	1
4309	1001	1405	2
4316	1001	1384	3
4317	1001	1347	4
4314	1001	1343	5
4307	1001	1321	6
4311	1001	1307	7
4310	1001	1280	8
4312	1001	1267	9
4313	1001	1254	10

27. Rerun the previous query, but instead of using `RANK()`, use `DENSE_RANK()`. What is the difference between the two functions?

`RANK` will skip ranks when there are ties. However, `DENSE_RANK` will always use the next available rank. For example, if there is a two-way tie for `total_amt_usd`, the next highest value will have a rank of 2 using `DENSE_RANK`, but a rank of 3 using `RANK`.

id	account_id	total	total_dense_rank
4308	1001	1410	1
4309	1001	1405	2
4316	1001	1384	3
4317	1001	1347	4
4314	1001	1343	5
4307	1001	1321	6
4311	1001	1307	7
4310	1001	1280	8
4312	1001	1267	9
4313	1001	1254	10

Aggregates in Window Functions

28. Select the `id`, `account_id`, and `standard_qty` variable from the orders table, then create a column called `dense_rank that` ranks this `standard_qty` amount of paper (ordered by month) for each account. In addition, create a `sum_std_qty` which gives you the running total for account (ordered by month). Repeat the last task to get the avg, min, and max.

```
SELECT id,
       account_id,
       standard_qty,
       DATE_TRUNC('month', occurred_at)               AS month,
       DENSE_RANK() OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS dense_rank,
       SUM(standard_qty) OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS sum_std_qty,
       COUNT(standard_qty) OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS count_std_qty,
       AVG(standard_qty) OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS avg_std_qty,
       MIN(standard_qty) OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS min_std_qty,
       MAX(standard_qty) OVER (
           PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)) AS max_std_qty
FROM orders
LIMIT 10
```

id	account_id	standard_qty	month	dense_rank	sum_std_qty	count_std_qty	avg_std_qty	min_std_qty	max_std_qty
1	1001	123	2015-10-01	1	123	1	123.0000	123	123
4307	1001	506	2015-11-01	2	819	3	273.0000	123	506
2	1001	190	2015-11-01	2	819	3	273.0000	123	506
3	1001	85	2015-12-01	3	1430	5	286.0000	85	526
4308	1001	526	2015-12-01	3	1430	5	286.0000	85	526
4309	1001	566	2016-01-01	4	2140	7	305.7143	85	566
4	1001	144	2016-01-01	4	2140	7	305.7143	85	566
4310	1001	473	2016-02-01	5	2721	9	302.3333	85	566
5	1001	108	2016-02-01	5	2721	9	302.3333	85	566
6	1001	103	2016-03-01	6	3322	11	302.0000	85	566

We can use the `WINDOW` keyword to provide an alias for our window functions. In the above query, we have to repeat

```
PARTITION BY account_id
           ORDER BY DATE_TRUNC('month', occurred_at)
```

many times. Instead, let's define it as an alias using the `WINDOW` keyword.

29. Give an alias for our window function in the previous question, and call it `account_month_window`.

```
SELECT id,
       account_id,
       standard_qty,
       date_trunc('month', occurred_at),
       dense_RANK() OVER account_month_window        AS dense_rank,
       sum(standard_qty) OVER account_month_window   AS sum_std_qty,
```

```
       count(standard_qty) OVER account_month_window AS count_std_qty,
       max(standard_qty) OVER account_month_window   AS max_std_qty,
       min(standard_qty) OVER account_month_window   AS min_std_qty
FROM orders
    WINDOW account_month_window AS (PARTITION BY account_id ORDER BY date_trunc('month', occurred_at))
LIMIT 10
```

Deduplication

30. There appear to be duplicate records in the `orders_duplicated` table. Deduplicate that table, ensuring that there is only one order ID.

For those of you who cannot connect to the remote databases and would like to create the `duplicated_orders` table from scratch, you can use the following DDL statement:

```
CREATE TABLE duplicated_orders AS

SELECT ROW_NUMBER() OVER (), duplicated.*
FROM (
         SELECT *, CURRENT_TIMESTAMP AS updated_at
         FROM orders
         UNION
         SELECT *, CURRENT_TIMESTAMP + INTERVAL '2 hours' AS updated_at
         FROM orders
         UNION
         SELECT *, CURRENT_TIMESTAMP + INTERVAL '23 hours' AS updated_at
         FROM orders) duplicated;
```

This will create a table called `duplicated_orders` with three copies of the `orders` data, all with one additional column called `updated_at`. This column will contain a timestamp for when the record was updated. One version is set to the current time when I inserted the rows (Sunday afternoon PST). The 2nd `updated_at` is two hours ahead. The third `updated_at` is 23 hours ahead.

Now, we only want to get the most recent record if there's duplicate orders. The first question is to identify duplicates and rank them by the `updated_at` time:

```
SELECT *,
       ROW_NUMBER() OVER
           (PARTITION BY id ORDER BY updated_at DESC) AS order_rank
FROM duplicated_orders
LIMIT 5
```

row_number	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd	updated_at	order_rank
12536	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	2020-11-02 16:21:18	1
9168	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	2020-11-01 19:21:18	2
14582	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	2020-11-01 17:21:18	3
8341	2	1001	2015-11-05 03:34:33	190	41	57	288	948.10	307.09	462.84	1718.03	2020-11-02 16:21:18	1
6804	2	1001	2015-11-05 03:34:33	190	41	57	288	948.10	307.09	462.84	1718.03	2020-11-01 19:21:18	2

Then, we want to get all of the rows that are not `order_rank = 1` - these are the rows that are going to "older" and should be removed. Remember, we only want to most recent row for the same order.

So we'll filter where `order_ranking > 1`:

```
SELECT *
FROM (
         SELECT *,
                ROW_NUMBER() OVER
                    (PARTITION BY id ORDER BY updated_at DESC) AS order_ranking
         FROM duplicated_orders) order_rankings
```

```
WHERE order_ranking > 1
LIMIT 5
```

row_number	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd	updated_at	order_ranking
9168	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	2020-11-01 19:21:18	2
14582	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	2020-11-01 17:21:18	3
6804	2	1001	2015-11-05 03:34:33	190	41	57	288	948.10	307.09	462.84	1718.03	2020-11-01 19:21:18	2
1158	2	1001	2015-11-05 03:34:33	190	41	57	288	948.10	307.09	462.84	1718.03	2020-11-01 17:21:18	3
294	3	1001	2015-12-04 04:21:55	85	47	0	132	424.15	352.03	0.00	776.18	2020-11-01 19:21:18	2

The result set returned here are all the order rows that are duplicated. So now we'll need to plug this into a `DELETE` statement. We'll grab the list of `row_numbers` that are duplicates, and then forward them along to the `WHERE` statement of our `DELETE`:

```
DELETE
FROM duplicated_orders
WHERE row_number IN (
    SELECT row_number -- get only the row numbers that are marked for deletion
    FROM (
             SELECT *, -- generate an order_ranking to determine which row is most recent
                    ROW_NUMBER() OVER
                        (PARTITION BY id ORDER BY updated_at DESC) AS order_ranking
             FROM duplicated_orders) order_rankings
    WHERE order_ranking > 1)
```

Percentiles

You can use window functions to identify what percentile (or quartile, or any other subdivision) a given row falls into. The syntax is `NTILE`(*# of buckets*). In this case, ORDER BY determines which column to use to determine the quartiles (or whatever number of tiles you specify).

In cases with relatively few rows in a window, the `NTILE` function doesn't calculate exactly as you might expect. For example, If you only had two records and you were measuring percentiles, you'd expect one record to define the 1st percentile, and the other record to define the 100th percentile. Using the `NTILE` function, what you'd actually see is one record in the 1st percentile, and one in the 2nd percentile.

In other words, when you use a `NTILE` function but the number of rows in the partition is less than the `NTILE`(number of groups), then `NTILE` will divide the rows into as many groups as there are members (rows) in the set but then stop short of the requested number of groups. If you're working with very small windows, keep this in mind and consider using quartiles or similarly small bands.

31. Use the `NTILE` functionality to divide the accounts into 4 levels in terms of the amount of standard_qty for their orders. Your resulting table should have the `account_id`, the `occurred_at` time for each order, the total amount of `standard_qty` paper purchased, and one of four levels in a standard_quartile column.

```
SELECT id,
       account_id,
       occurred_at,
       standard_qty,
       NTILE(4) OVER (PARTITION BY account_id ORDER BY standard_qty) AS standard_quartile
  FROM orders
 ORDER BY account_id DESC
limit 10
```

id	account_id	occurred_at	standard_qty	standard_quartile
6912	4501	2016-12-21 13:30:42	61	2
4301	4501	2016-07-29 20:06:39	111	3
6908	4501	2016-06-29 04:03:39	11	1
6910	4501	2016-08-27 00:58:11	16	2
6911	4501	2016-11-22 06:52:22	63	2
4300	4501	2016-06-29 03:57:11	104	3
6909	4501	2016-07-29 19:58:32	5	1
4305	4501	2016-11-22 06:57:04	6	1
4299	4501	2016-05-30 04:18:34	15	1
4306	4501	2016-12-21 13:43:26	126	3

32. **YOUR TURN**- Use the `NTILE` functionality to divide the accounts into two levels in terms of the amount of gloss_qty for their orders. Your resulting table should have the account_id, the occurred_at time for each order, the total amount of `gloss_qty` paper purchased, and one of two levels in a `gloss_half` column.

id	account_id	occurred_at	gloss_qty	gloss_half
4299	4501	2016-05-30 04:18:34	11	1
6911	4501	2016-11-22 06:52:22	67	2
4303	4501	2016-09-25 01:44:03	0	1
4302	4501	2016-08-27 00:48:17	11	1
4300	4501	2016-06-29 03:57:11	14	1
4301	4501	2016-07-29 20:06:39	16	2
4305	4501	2016-11-22 06:57:04	0	1
4306	4501	2016-12-21 13:43:26	0	1
4304	4501	2016-10-24 08:50:37	6	1
6909	4501	2016-07-29 19:58:32	91	2

33. **YOUR TURN** - Use the `NTILE` functionality to divide the orders for each account into 100 levels in terms of the amount of `total_amt_usd` for their orders. Your resulting table should have the account_id, the occurred_at time for each order, the total amount of `total_amt_usd` paper purchased, and one of 100 levels in a **total_percentile** column.

id	account_id	occurred_at	total_amt_usd	total_percentile
4304	4501	2016-10-24 08:50:37	1122.55	6
6910	4501	2016-08-27 00:58:11	1449.74	9
4306	4501	2016-12-21 13:43:26	628.74	3
4301	4501	2016-07-29 20:06:39	974.17	5
4302	4501	2016-08-27 00:48:17	1175.47	7
4303	4501	2016-09-25 01:44:03	1324.34	8
4305	4501	2016-11-22 06:57:04	86.78	1
4299	4501	2016-05-30 04:18:34	157.24	2
4300	4501	2016-06-29 03:57:11	875.54	4
6911	4501	2016-11-22 06:52:22	1473.92	10