Module 1: Path & Pattern Analysis

Teradata Vantage Analytics Workshop

BASIC

Objectives

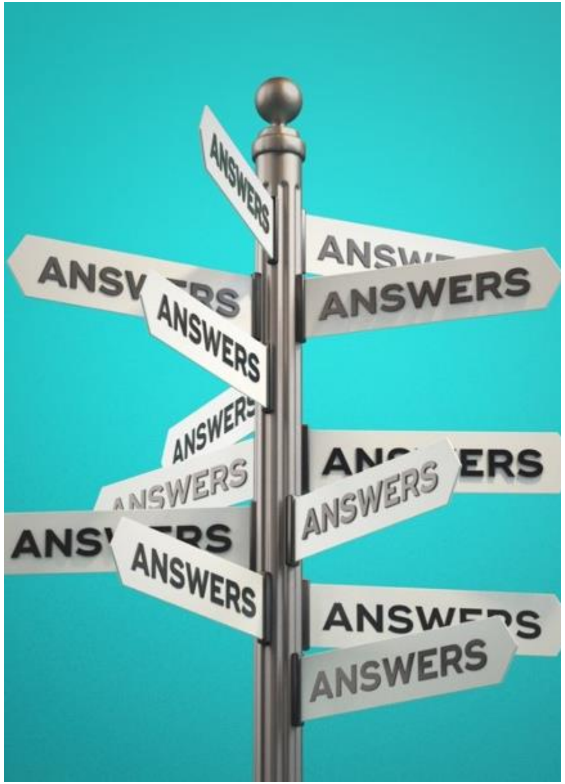
After completing this module, you will be able to:

- Describe what the Sessionize and Attribution functions do
- Describe typical use cases for Sessionize and Attribution
- Write Sessionize and Attribution queries
- Interpret the output of Sessionize and Attribution queries

For more info go to docs.teradata.com click Teradata Vantage, download: Teradata Vantage Analytic Function Reference Guide.

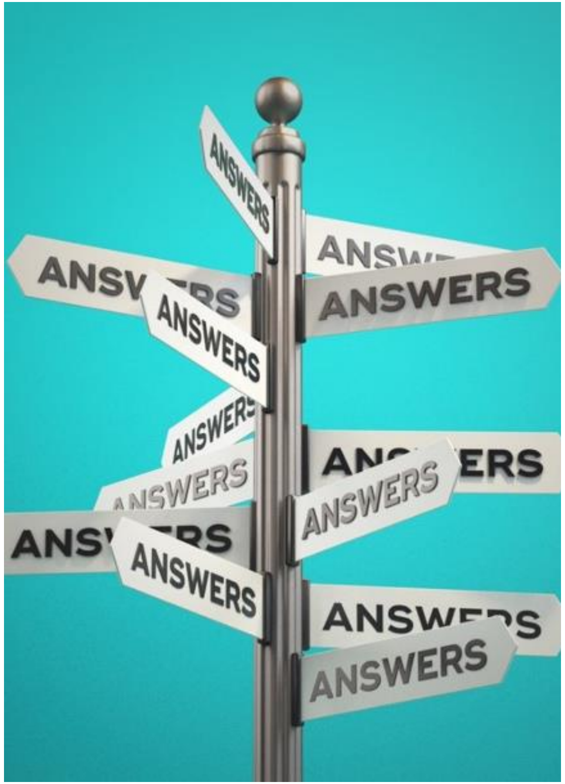
Topics

- Sessionize
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - Labs
  - Review
- Attribution
  - Background Information (Description and Use Cases)
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - Review



Current Topic – Sessionize Background Information

- **Sessionize**
  - **Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)**
  - Labs
  - Review
- Attribution
  - Background Information (Description and Use Cases)
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - Review



Sessionize Description

- The **Sessionize** function maps each 'event' (such as a web click) in a session to a unique session identifier
- A 'session' is defined as a sequence of 'events' by one user that are separated by at most n seconds
- The function is useful both for sessionization and for detecting web crawler (robot) activity
- It is typically used to understand user browsing behavior on a web site

Sessionize Use Case Examples

- A Retailer wishes to know which pages on its website are visited most often
- A Banking institution wishes to know if there have been any attempted robot infiltrations into customer accounts
- A Social-media website wishes to sell advertising space and wants to know the number of sessions each user has per day, and the average length in time of those sessions

Other examples:

What's AVG number of Sessions before new Customers start declining in Sales revenue? I'll start Marketing campaign prior to this to minimize this behavior

Is Number of Visits seasonal? If so, ensure Advertisements focus on strongest month when Number of Visits is Maximum

Sessionize Workflow

Input Table → Sessionize → Console or Output Table

- The **Sessionize** function reads data from an input table, view, or query, and then outputs **sessionid** (per specified arguments)
- For example, if a **userid** has 2 consecutive clicks within 1 minute of each other, consider that the same 'session'
- If > 1 minute, then increment **sessionid** counter by 1

The **Sessionize** function outputs a **sessionid** column. Note that **sessionid** always begins at **0** within each new partition

Input

timestamp	userid	…
10:00:00	10	…
00:58:24	76	
10:00:24	10	
02:30:33	76	
10:01:23	10	
10:02:40	10	

Output

timestamp	userid	…	sessionid
10:00:00	10	…	0
10:00:24	10		0
10:03:00	10		1
10:05:30	10		2
00:59:24	76		0
02:30:33	76		1

Userid 10 has three 'sessions': 0, 1, and 2

Userid 76 has two 'sessions': 0 and 1

Sessionize Syntax


```
SELECT * FROM Sessionize
(ON { table | view | (query) }
PARTITION BY expression [,...]
ORDER BY order_column
USING
TimeColumn ('timestamp_column')
TimeOut ('session_timeout')
[ ClickLag ('min_human_click_lag') ]
[ EmitNull ({'true'|'t'|'yes'|'y'|'1'|'false'|'f'|'no'|'n'|'0'})]
) as alias;
```

Sessionize Required Arguments

- **TimeColumn**: Specify the name of the input column that contains the click times. **Note:** The timestamp_column must also be an order_column
- **TimeOut**: Specify the number of seconds at which the session times out. If session_timeout seconds elapse after a click, the next click starts a new session. The data type of session_timeout is DOUBLE PRECISION

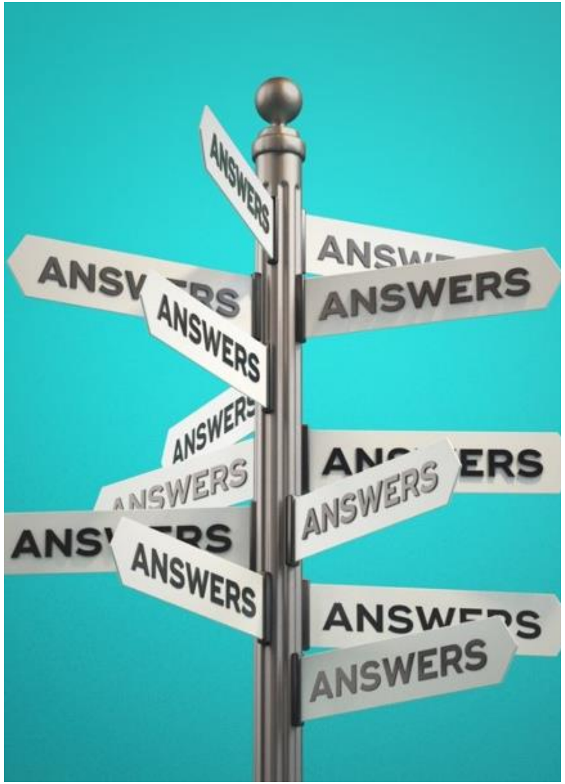
Sessionize Optional Arguments

- **ClickLag** [Optional]: Specify the minimum number of seconds between clicks for the session user to be considered human. If clicks are more frequent, indicating that the user is a bot, the function ignores the session. The min_human_click_lag must be less than session_timout. The data type of min_human_click_lag is DOUBLE PRECISION. Default behavior: The function ignores no session, regardless of click frequency

- **EmitNull** [Optional]: Specify whether to output rows that have NULL values in their session id and ClickLag columns, even if their timestamp_column has a NULL value. Default: 'false'

Current Topic – Sessionize Labs

- **Sessionize**
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - **Labs**
  - Review
- Attribution
  - Background Information (Description and Use Cases)
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - Review



Lab 1a: View the Data

Goal: **Sessionize** below data to count how many visits each userid had to a website

```
SELECT * FROM sessionme;
```

	userid	clicktime	productid	pagetype	referrer	price
1	578	2018-01-05 11:10:00.000000	bose	checkout		750.00
2	333	2018-01-05 09:10:00.000000		home	www.yahoo.com	.00
3	578	2018-01-05 11:11:00.000000		home	www.godaddy.com	.00
4	333	2018-01-05 09:13:00.000000		home	www.google.com	.00
5	578	2018-01-05 11:14:00.000000	iphone	checkout		650.00
6	333	2018-01-05 09:14:00.000000		home	www.google.com	.00
7	578	2018-01-05 11:12:00.000000	bose	checkout		340.00
8	333	2018-01-05 10:06:00.000000		home	www.google.com	.00
9	578	2018-01-05 11:13:00.000000	ipad	checkout		450.00
10	333	2018-01-05 10:07:00.000000		home		.00
11	333	2018-01-05 10:08:00.000000		home		.00
12	333	2018-01-05 10:09:00.000000	iphone	checkout		650.00
13	333	2018-01-05 09:11:00.000000	ipod	checkout	www.yahoo.com	200.20
14	333	2018-01-05 09:12:00.000000	bose	checkout		340.00

Lab 1b: Sessionize

Query: Sessionize User's clicks that are within 1 minute of each other. Starting at SESSIONID = 0, if current row's time is <= 1 min, it's same SESSIONID. Otherwise increment by 1

```
SELECT * FROM Sessionize
(ON sessionme
 PARTITION BY userid
 ORDER BY clicktime
 USING
 TimeColumn ('clicktime')
 TimeOut (60)
 ClickLag (0.2)
 EmitNull ('false')
) ORDER BY userid, clicktime;
```

2 new columns created

Output

userid	clicktime	productid	pagetype	referrer	price	SESSIONID	CLICKLAG
333	2018-01-05 09:10:00.000000		home	www.yahoo.com	.00	0	f
333	2018-01-05 09:11:00.000000	ipod	checkout	www.yahoo.com	200.20	0	f
333	2018-01-05 09:12:00.000000	bose	checkout		340.00	0	f
333	2018-01-05 09:13:00.000000		home	www.google.com	.00	0	f
333	2018-01-05 09:14:00.000000		home	www.google.com	.00	0	f
333	2018-01-05 10:06:00.000000		home	www.google.com	.00	1	f
333	2018-01-05 10:07:00.000000		home		.00	1	f
333	2018-01-05 10:08:00.000000		home		.00	1	f
333	2018-01-05 10:09:00.000000	iphone	checkout		650.00	1	f
578	2018-01-05 11:10:00.000000	bose	checkout		750.00	0	f
578	2018-01-05 11:11:00.000000		home	www.godaddy.com	.00	0	f
578	2018-01-05 11:12:00.000000	bose	checkout		340.00	0	f
578	2018-01-05 11:13:00.000000	ipad	checkout		450.00	0	f
578	2018-01-05 11:14:00.000000	iphone	checkout		650.00	0	f

Note the following:

- **userid 333** had two visits within the Partition, denoted by **SESSIONID** values **0** and **1**
- **userid 578** had one visit within the Partition, denoted by **SESSIONID** value **0**

Lab 2a: View the Data

Here, we view the **bank_web** table. The next many slides will walk through the data and various examples of using the **Sessionize** syntax against this table

ANSI SQL

```
SELECT * FROM bank_web
WHERE customer_id IN (620, 8263, 30324)
ORDER BY customer_id, datestamp;
```

	customer_id	page	datestamp
1	620	ACCOUNT SUMMARY	2004-03-20 12:35:46.000000
2	620	ACCOUNT HISTORY	2004-03-20 12:38:56.000000
3	620	ACCOUNT HISTORY	2004-03-20 12:41:29.000000
4	620	PROFILE UPDATE	2004-03-20 12:42:51.000000
5	620	VIEW DEPOSIT DETA...	2004-03-20 12:45:10.000000
6	8263	ACCOUNT SUMMARY	2004-03-21 20:38:54.000000
7	8263	ACCOUNT HISTORY	2004-03-21 20:42:39.000000
8	8263	PROFILE UPDATE	2004-03-21 20:44:59.000000
9	8263	FUNDS TRANSFER	2004-03-21 20:46:04.000000
10	30324	ACCOUNT SUMMARY	2004-05-01 15:03:13.000000
11	30324	ONLINE STATEMENT ...	2004-05-01 15:07:06.000000
12	30324	ACCOUNT SUMMARY	2004-05-01 15:10:43.000000
13	30324	FAQ	2004-05-01 15:12:12.000000
14	30324	ACCOUNT HISTORY	2004-05-01 15:15:15.000000

Lab 2b: Required Arguments and Output


```
SELECT * FROM Sessionize
(ON bank_web
 PARTITION BY customer_id
 ORDER BY datestamp
 USING
 TimeColumn ('datestamp')
 TimeOut (600)
) ORDER BY customer_id,
           datestamp;
```

- The **ON** clause contains the input table
- The **PARTITION BY** argument specifies for each distinct instance of **customer_id**, the sessionize function will re-start at a value of **0**
- The **ORDER BY** argument specifies that for each customer, data will be sessionized according to the **datestamp** value (ascending by default)
- The **USING** clause defines the **TimeColumn** (the input column that contains our timestamp data) and the user-defined **TimeOut** value (must be defined in seconds). As long as a **current row occurs within 600 seconds of prior row**, they will be considered as part of the same 'session'

Lab 2b: Required Arguments and Output (cont.)

Customer_id = 32 had 6 visits over 3-day span

	customer_id	page	datestamp	SESSIONID
1	32	ACCOUNT SUMMARY	2004-04-14 20:57:15.000000	0
2	32	VIEW DEPOSIT DETA...	2004-04-14 21:01:07.000000	0
3	32	ACCOUNT SUMMARY	2004-04-15 10:24:53.000000	1
4	32	FUNDS TRANSFER	2004-04-15 10:28:43.000000	1
5	32	BILL MANAGER FORM	2004-04-15 10:29:20.000000	1
6	32	CUSTOMER SUPPORT	2004-04-15 10:29:54.000000	1
7	32	ACCOUNT SUMMARY	2004-04-15 15:19:29.000000	2
8	32	ACCOUNT HISTORY	2004-04-15 15:20:20.000000	2
9	32	FUNDS TRANSFER	2004-04-15 15:24:19.000000	2
10	32	ACCOUNT SUMMARY	2004-04-15 21:50:04.000000	3
11	32	VIEW DEPOSIT DETA...	2004-04-15 21:53:19.000000	3
12	32	CUSTOMER SUPPORT	2004-04-15 21:54:10.000000	3
13	32	FUNDS TRANSFER	2004-04-15 21:54:58.000000	3
14	32	ACCOUNT SUMMARY	2004-04-16 14:18:14.000000	4
15	32	BILL MANAGER FORM	2004-04-16 14:20:34.000000	4
16	32	BILL MANAGER ENR...	2004-04-16 14:22:52.000000	4
17	32	FAQ	2004-04-16 14:25:34.000000	4
18	32	ACCOUNT SUMMARY	2004-04-16 14:28:09.000000	4
19	32	ACCOUNT SUMMARY	2004-04-16 17:49:32.000000	5
20	32	FUNDS TRANSFER	2004-04-16 17:51:04.000000	5
21	32	FUNDS TRANSFER	2004-04-16 17:54:33.000000	5
22	32	FUNDS TRANSFER	2004-04-16 17:55:10.000000	5
23	32	ACCOUNT HISTORY	2004-04-16 17:58:56.000000	5

- Here, we are viewing the output of our query from the previous page
- Note the creation of the **SESSIONID** column
- As long as the clicks of a single **customer_id** occurred **within 600 seconds** of one another, they will share the same **SESSIONID** value

Lab 3a: Specifying a Query in the ON Clause


```
SELECT * FROM Sessionize
(ON (SELECT * FROM bank_web
     WHERE customer_id
     IN (620, 8263, 30324))
 PARTITION BY customer_id
 ORDER BY datestamp
 USING
 TimeColumn ('datestamp')
 TimeOut (120)
) ORDER BY customer_id, datestamp;
```

- Note that you can also specify a query in the **ON** clause to select desired input data, as opposed to just putting the name of a table or view that contains the input data (as we did in the previous lab)
- When specifying a query, you must enclose it within parentheses
- If desired, you could write your query to **SELECT** only <u>certain columns to increase performance too</u>

Lab 3a: Specifying a Query in the ON Clause (cont.)

For each **customer_id**, as long as clicks occur within 120 seconds of one another, they will be part of the same **SESSIONID**

	customer_id	page	datestamp	SESSIONID
1	620	ACCOUNT SUMMARY	2004-03-20 12:35:46.000000	0
2	620	ACCOUNT HISTORY	2004-03-20 12:38:56.000000	1
3	620	ACCOUNT HISTORY	2004-03-20 12:41:29.000000	2
4	620	PROFILE UPDATE	2004-03-20 12:42:51.000000	2
5	620	VIEW DEPOSIT DETAILS	2004-03-20 12:45:10.000000	3
6	8263	ACCOUNT SUMMARY	2004-03-21 20:38:54.000000	0
7	8263	ACCOUNT HISTORY	2004-03-21 20:42:39.000000	1
8	8263	PROFILE UPDATE	2004-03-21 20:44:59.000000	2
9	8263	FUNDS TRANSFER	2004-03-21 20:46:04.000000	2
10	30324	ACCOUNT SUMMARY	2004-05-01 15:03:13.000000	0
11	30324	ONLINE STATEMENT ENROLLMENT	2004-05-01 15:07:06.000000	1
12	30324	ACCOUNT SUMMARY	2004-05-01 15:10:43.000000	2
13	30324	FAQ	2004-05-01 15:12:12.000000	2
14	30324	ACCOUNT HISTORY	2004-05-01 15:15:15.000000	3

Lab 4: Detecting Robots

```
SELECT * FROM Sessionize
(ON (SELECT * FROM bank_clicks
     WHERE customer_id IN (7172))
 PARTITION BY customer_id
 ORDER BY datestamp
 USING
 TimeColumn ('datestamp')
 TimeOut (60)
 ClickLag (0.1)
) ORDER BY customer_id, datestamp;
```

- We can use the optional argument **ClickLag** to detect possible bot activity
- Just like **Timeout**, **ClickLag** is expressed in seconds
- Any clicks that occur within **0.1** seconds of one another will be flagged accordingly in the output

Query: Customer 7172 can't login to their on-line bank account.

Write query that will SESSIONIZE the bank_clicks table for customer_id = 7172 with a TIMEOUT = 60 seconds and robot ClickLag = 0.10

Does anything look fishy?

Lab 4: Detecting Robots (cont.)

customer_id	page	datestamp	SESSIONID	CLICKLAG
7172	ACCOUNT SUMMARY	2004-03-22 04:46:12.0…	0	f
7172	FUNDS TRANSFER	2004-03-22 04:48:40.0…	1	f
7172	FAQ	2004-03-22 04:50:11.0…	2	f
7172	FUNDS TRANSFER	2004-03-22 04:53:43.0…	3	f
7172	VIEW DEPOSIT DETA…	2004-03-22 04:57:39.0…	4	f
7172	PROFILE UPDATE	2004-03-22 05:01:33.0…	5	f
7172	VIEW DEPOSIT DETA…	2004-03-23 11:19:37.0…	41	f
7172	VIEW DEPOSIT DETA…	2004-03-23 11:23:16.0…	42	f
7172	ACCOUNT SUMMARY	2004-03-23 20:28:33.0…	43	f
7172	BILL MANAGER	2004-03-23 20:29:18.0…	43	f
7172	VIEW DEPOSIT DETA…	2004-03-23 20:32:32.0…	44	f
7172	FUNDS TRANSFER	2004-03-23 20:33:34.0…	45	f
7172	VIEW DEPOSIT DETA…	2004-03-23 20:34:46.0…	46	f
7172	VIEW DEPOSIT DETA…	2004-03-23 20:36:59.0…	47	f
7172	FAQ	2004-03-23 20:38:07.0…	48	f
7172	LOGIN	2014-03-25 04:00:00.0…	49	f
7172	LOGIN	2014-03-25 04:00:00.1…	49	t
7172	LOGIN	2014-03-25 04:00:00.2…	49	t
7172	LOCKOUT	2014-03-25 04:00:00.3…	49	t

- The **CLICKLAG** column receives a value of '**t**' if a click occurred within **0.1 seconds** of the previous click
- For **SESSIONID 49**, it appears that a bot was attempting to log into the customer's bank account before being locked out

Lab 5a: Create Sessionized Data


```
CREATE SET TABLE chips_sessionized AS
(SELECT * FROM Sessionize
(ON (SELECT remote_host, request_time,
      requested_page
     FROM chips_clean)
 PARTITION BY remote_host
 ORDER BY request_time asc
 USING
 TimeColumn ('request_time')
 TimeOut (3600)
)
)
WITH DATA;
```

- It will often be beneficial to land your **Sessionize** results into a physical table for further analysis and/or to serve as an input into other Teradata VANTAGE functions, such as **nPath**
- Here, we are 'sessionizing' a subset of columns from the **chips_clean** table
- Each 'session' is defined as clicks made within the same window of **3,600 seconds** (one hour)

Lab 5b: View the Sessionized Data

Source Data

- chips_clean
  - Columns
    - remote_host [VARCHAR (1000), Nullable, PI]
    - remote_log_name [VARCHAR (1000), Nullable]
    - remote_user [VARCHAR (1000), Nullable]
    - request_time [TIMESTAMP, Nullable, PI]
    - requested_page [VARCHAR (1000), Nullable]
    - final_status [VARCHAR (1000), Nullable]
    - bytes_sent_CLF [VARCHAR (1000), Nullable]
    - referrer [VARCHAR (1000), Nullable]
    - request:User-agent [VARCHAR (1000), Nullable]

Sessionized Data

- chips_sessionized
  - Columns
    - remote_host [VARCHAR (1000), Nullable, PI]
    - request_time [TIMESTAMP, Nullable]
    - requested_page [VARCHAR (1000), Nullable]
    - SESSIONID [INTEGER, Nullable]

```
SELECT * FROM chips_sessionized;
```

	remote_host	request_time	requested_page	SESSIONID
1	136.243.36.89	2015-01-30 10:50:33.000000	/events.php?year=2058&month=8	0
2	125.209.235.171	2015-02-04 11:29:08.000000	/about.php	13
3	136.243.36.89	2015-02-21 23:43:23.000000	/events.php?year=2107&month=5	3
4	210.23.82.12	2015-01-26 15:58:45.000000	/products.php?cid=1	0
5	162.243.194.124	2015-02-08 10:27:18.000000	/events.php?year=2031&month=11	0

Lab 5c: View Most Popular Pages

Here, we are using our sessionization results to discover **Most Popular Pages** on our website; i.e., those visited in the greatest number of sessions

ANSI SQL

```
SELECT requested_page,
COUNT (DISTINCT remote_host || '_ '
|| sessionid) AS distinct_sessions
FROM chips_sessionized
GROUP BY requested_page
HAVING distinct_sessions >= 700
ORDER BY distinct_sessions DESC;
```

	requested_page	distinct_sessions
1	/products.php?cid=1	2422
2	/products.php?cid=6	1868
3	/contact.php	1771
4	/about.php	1469
5	/product.php?pid=34	1124
6	/cart.php	986
7	/products.php	977
8	/glutenfree.php	823
9	/locator.php	745
10	/account.php	744

Lab 5d: Create Summary Table

- Here, we have created a table comprised of one row per **remote_host**, **SESSIONID**
- We have populated columns to specify general metrics about each session
- We will use this data to answer questions such as the following:
  - How many pages visited per session?
  - How many distinct pages visited per session?
  - How long in duration is each session?
  - What % of sessions contain an actual order?

You will be running a series of CREATE TABLE statements within Teradata Studio

remote_host	SESSIONID	checkouts	payments	pages	distinct_pages	min_request_time	max_request_time	session_duration
72.8.191.4	0	0	0	1	1	2015-01-05 23:34:27.000000	2015-01-05 23:34:27.000000	0 00:00:00.000000
107.144.135.28	0	1	0	9	7	2015-01-05 18:10:41.000000	2015-01-05 18:18:23.000000	0 00:07:42.000000
157.55.39.164	96	0	0	2	2	2015-02-01 09:05:51.000000	2015-02-01 09:06:01.000000	0 00:00:10.000000
66.249.67.28	39	0	0	1	1	2015-02-07 01:14:33.000000	2015-02-07 01:14:33.000000	0 00:00:00.000000
66.249.69.148	15	0	0	1	1	2015-01-04 19:08:55.000000	2015-01-04 19:08:55.000000	0 00:00:00.000000
66.249.69.167	111	0	0	3	3	2015-01-16 02:30:33.000000	2015-01-16 03:31:14.000000	0 01:00:41.000000
188.165.15.27	0	0	0	1	1	2015-01-16 23:58:07.000000	2015-01-16 23:58:07.000000	0 00:00:00.000000
54.85.40.134	44	0	0	1	1	2015-01-24 20:36:06.000000	2015-01-24 20:36:06.000000	0 00:00:00.000000
157.55.39.110	53	0	0	2	2	2015-01-19 11:14:37.000000	2015-01-19 11:51:01.000000	0 00:36:24.000000
66.249.65.193	0	0	0	1	1	2015-01-29 05:14:27.000000	2015-01-29 05:14:27.000000	0 00:00:00.000000

Lab 5e: Retrieve General Session Metrics

Here, we Summarized all session data to display average metrics in aggregate

```
SELECT count (distinct remote_host) as remote_hosts,
       count (distinct remote_host || '_' || sessionid) as sessions,
       sessions*1.00 / remote_hosts as avg_sessions_per_host,
       cast (avg (pages) as decimal (4,2)) as avg_pages,
       cast (avg (distinct_pages) as decimal (4,2)) as avg_distinct_pages,
       avg (session_duration) as avg_session_duration
FROM x_summary;
```

Output

remote_hosts	sessions	avg_sessions_per_host	avg_pages	avg_distinct_pages	avg_session_duration
7969	20278	2.54	3.63	3.19	0 00:14:24.948614

Note: Your Output may differ than results here

Lab 5f: Are There Abandoned Carts?

- Here, we have written a query to identify the number of sessions that included Checkout and Payment

```
SELECT case when payments > 0 then 'y' else 'n' end as sessions_with_payment,
       case when checkouts > 0 then 'y' else 'n' end as sessions_with_checkout,
       count (*) as number_of_sessions
FROM x_summary
GROUP BY sessions_with_payment,sessions_with_checkout
ORDER BY sessions_with_payment,sessions_with_checkout;
```

- Note that only a tiny fraction of sessions included a Payment
- There is a problem with 'Abandoned Carts'

	sessions_with_payment	sessions_with_checkout	number_of_sessions
1	n	n	19953
2	n	y	233
3	y	y	92

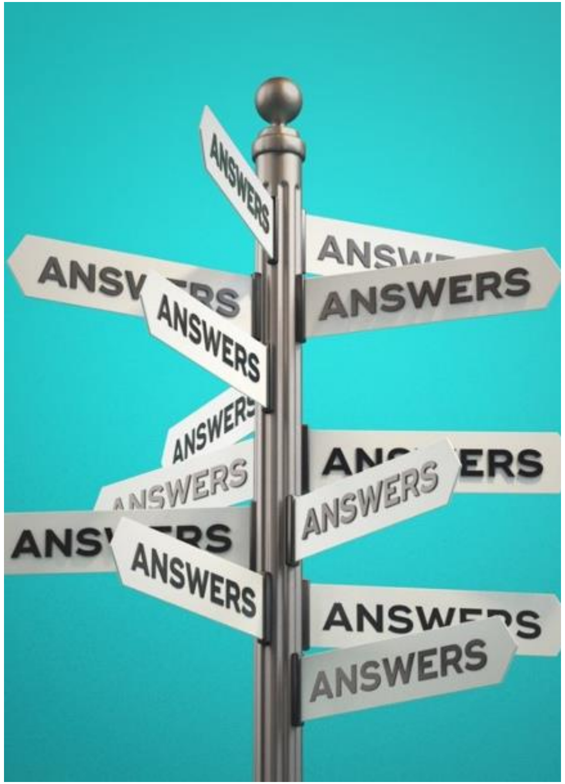
Lab 5g: General Session Metrics by Checkout/Purchase Groups

- Here, we have written a query to display general metrics parsed out by whether the session included checkout and/or payment (or not)
- Note the following:
  - Few customers make purchases
  - Abandoned carts are a problem
  - People who make purchases tend to be more engaged with the website (visit more pages and revisit pages already visited)
  - Precious few customers who actually make a purchase do so more than once

	sessions_with_payment	sessions_with_checkout	remote_hosts	sessions	avg_sessions_per_host	avg_pages	avg_distinct_pages	avg_session_duration
1	n	n	7739	19953	2.58	3.53	3.14	0 00:14:28.832657
2	n	y	222	233	1.05	8.68	5.74	0 00:09:49.793991
3	y	y	88	92	1.05	11.71	7.46	0 00:11:59.434783

Current Topic – Sessionize Review

- **Sessionize**
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - Labs
  - **Review**
- Attribution
  - Background Information (Description and Use Cases)
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - Review



Your Turn: Fix the Syntax ERRORs

The following code examples have bad syntax. Fix the code so it runs successfully

```
SELECT * FROM Sessionize
(ON sessionme
 PARTITION BY userid
 ORDER BY clicktime
 USING
 TimeColumns ('clicktime')
 TimeOut (60)
 ClickLag (0.2)
 EmitNull ('false')
) ORDER BY userid, clicktime;
```

```
SELECT * FROM Sessionize
(ON bank_clicks
 PARTITION BY customer_id
 ORDER BY datestamp
 USING
 TimeColumn ('datestamp')
 TimeOut ('600')
) ORDER BY customer_id, datestamp;
```

TimeColumn , not TimeColumns

TimeOut(600) , not TimeOut('600')

Your Turn: Fill in the Blanks (1st example only)

Fill in the five parts of the code (with ???? marks), so the code will run as expected

```
SELECT * FROM Sessionize
(ON TRNG_TDU_TD01.sessionme
 ????????? ?? userid
 ORDER BY clicktime
 USING
 ?????????? ('clicktime')
 ??????? (60)
 ???????? (0.2)
 ???????? ('false'))
ORDER BY userid, clicktime;
```

```
SELECT * FROM Sessionize
(ON TRNG_TDU_TD01.sessionme
 PARTITION BY userid
 ORDER BY clicktime
 USING
 TimeColumn ('clicktime')
 TimeOut (60)
 ClickLag (0.2)
 EmitNull ('false'))
ORDER BY userid, clicktime;
```

Sessionize Summary

In this module, you learned how to:

- Describe what the **Sessionize** function does
- Describe typical use cases for **Sessionize**
- Write **Sessionize** queries
- Interpret the output of **Sessionize** queries

Current Topic – Attribution Background Information

- Sessionize
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - Labs
  - Review
- **Attribution**
  - **Background Information (Description and Use Cases)**
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - Review

Description

- The **Attribution** function is typically used in web page analysis, where it lets companies assign weights to pages before certain events, such as buying a product
- Calculates attributions with a choice of distribution models and has two versions
  - **Multiple-Input Attribution:** Accepts <u>one or more input tables</u> and gets many parameters from other dimension tables. Recommended for large numbers of parameters. You must create tables of parameters, but can use the tables whenever you call the function, instead of specifying each parameter in an argument (Conversion and Model tables use **DIMENSION** keyword)

Description (cont.)

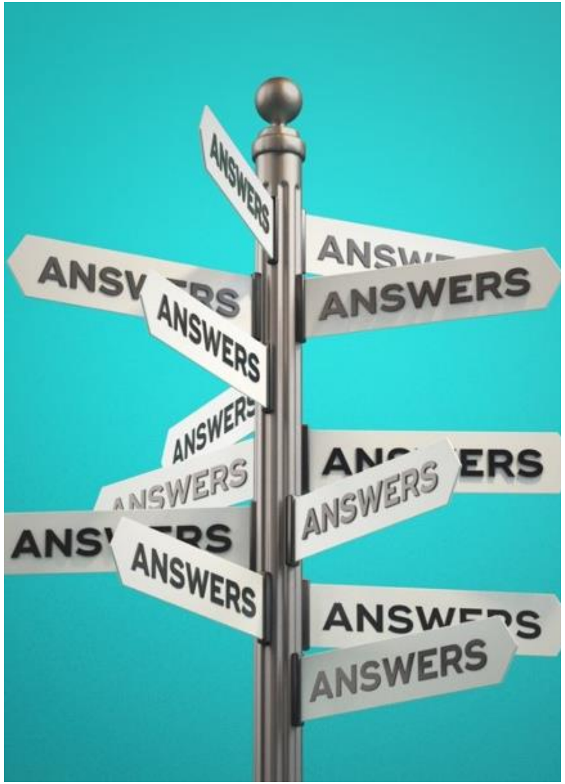
- The **Attribution** function can be coded to assign weights in the following fashions to events or actions that lead up to a 'Conversion event':
  - **LAST_CLICK, FIRST_CLICK, UNIFORM, WEIGHTED, EXPONENTIAL**
- It is the user of the function who decides *which* and *to what extent* antecedent events may or may not influence the existence of the conversion event; i.e., there is no black box or complex mathematical algorithm that determines *cause and effect* between events
  - If the conversion event is 'purchase product', and the user specifies that the five previous events to 'purchase product' should receive uniform credit for 'influencing' or 'causing' the 'purchase product' event, then each of the five previous events will receive 20% of the **Attribution**

Use Cases – Examples

- A Retailer wishes to know which pages on its website ultimately 'influence' or lead up to a product purchase
- A Telecommunications company wishes to understand for plan cancellations, which antecedent events are attributable to this
- A Manufacturer wants to analyze machine sensor data over time to understand *cause and effect* relationships between antecedent events and parts' failure or breakdown
- A Marketing department wishes to calculate the contribution of each touchpoint to a conversion (i.e., a 'sale'). With this knowledge, the Marketing department can plan out next year's budget with an emphasis on the highest-contributing touchpoints towards sales

Current Topic – Attribution Multiple-Input Models

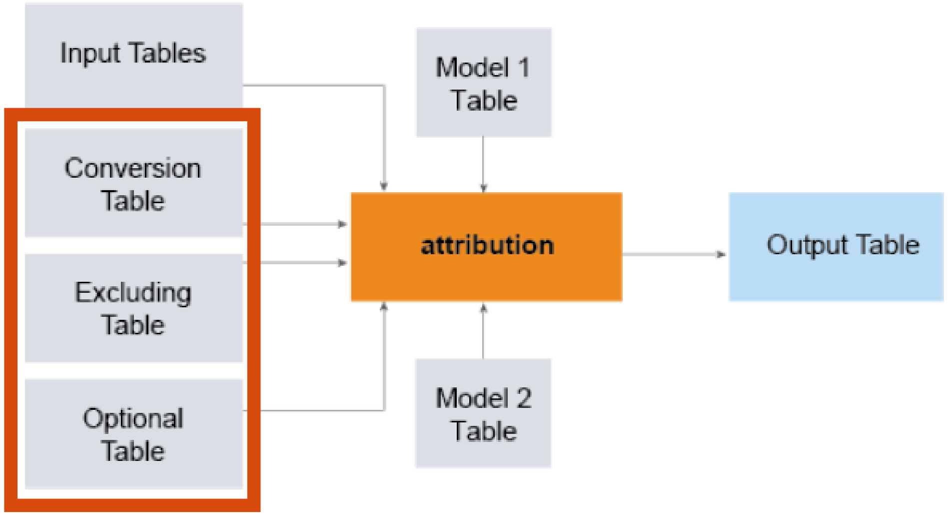
- Sessionize
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - Labs
  - Review
- **Attribution**
  - Background Information (Description and Use Cases)
  - **Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)**
  - Review



Workflow for Multiple-Input (Table) Models

Multiple-input models

- Input data is obtained from **<u>Multiple input</u>** and '**metadata**' tables
- Arguments are specified by referencing the contents of the various 'metadata' tables



Attribution Syntax (Multiple-Input Table Model)


```
SELECT * FROM Attribution
(ON { table | view | (query) } PARTITION BY user_id ORDER BY timestamp_column
[ ON { table | view | (query) } PARTITION BY user_id ORDER BY
timestamp_column ] [,...]
ON conversion_event_table AS conversion DIMENSION
[ ON excluding_event_table AS excluding DIMENSION ]
[ ON optional_event_table AS optional DIMENSION ]
ON model1_table AS model1 DIMENSION
[ ON model2_table AS model2 DIMENSION ]
USING
EventColumn ('event_column')
TimestampColumn ('timestamp_column')
WindowSize ({ 'rows:K' | 'seconds:K' | 'rows:K&seconds:K2' })
) AS alias ORDER BY user_id,time_stamp;
```

Attribution Required Arguments

- **EventColumn**: Specify the name of the input column that contains the clickstream events
- **Conversion Table**: Specify the conversion events table. Each conversion_event is a string or integer. Can have multiple events
- **TimestampColumn**: Specify the name of the input column that contains the timestamps of the clickstream events

teradata.

Attribution Required Arguments (cont.)

- **WindowSize:** Specify how to determine the maximum window size for the attribution calculation:
  - **rows:K** Consider maximum number of events to attribute, excluding events of types specified in excluding_event_table (or argument), which means assigning attributions to at most K effective events before current impact event
  - **seconds:K** Consider maximum time difference between current impact event and earliest effective event to attribute
  - **rows:K&seconds:K2** Consider both constraints and comply with stricter one

Attribution Required Arguments (cont.)

Model1 Table: Specify the Model Type in a table.
For example:

```
Model1 ('EVENT_REGULAR',
        'email:0.20:LAST_CLICK:NA','storePromo:0.80:WEIGHTED:0.4,0.3,0.2,0.1')
```

Here we are saying:

- Model type = 'EVENT_REGULAR' allows user to select just 'Touchpoint' values to score
- 'email' Touchpoints get 20% of the weight
- 'storePromo' Touchpoints get 80% of the weight
- Assuming 4 have 'storePromo', closest one (based on time to Conversion) gets weighted more heavily (40% of 80%) than the farther away 'storePromo' rows
- No other Touchpoints will be scored

Multiple-Input Model: Create Conversion_Event_Table


```
CREATE MULTISET TABLE
conversion_event_table
(conversion_events varchar (255));

INSERT INTO conversion_event_table
(conversion_events) values ('buy');

SELECT * FROM
conversion_event_table;
```

- Here, we are building, populating, and selecting from the required table that houses our conversion event of **buy**
- If we had multiple **conversion events**, they would each occupy their own row

conversion_event_table
Columns
conversion_events [VARCHAR (255), Nullable, PI]

conversion_events
Buy

Multiple-Input Model: Create Model_Table

Syntax

```
CREATE MULTISET TABLE model_table
(id integer, model varchar (255));
insert into model_table (id, model)
values (0, 'SEGMENT_ROWS');
insert into model_table (id, model)
values (1, '3:0.4:UNIFORM:NA');
insert into model_table (id, model)
values (2, '3:0.3:LAST_CLICK:NA');
insert into model_table (id, model)
values (3, '3:0.2:EXPONENTIAL:0.5,ROW');
insert into model_table (id, model)
values (4, '3:0.1:FIRST_CLICK:NA');
SELECT * FROM model_table ORDER BY id;
```

- Here, we are building, populating, and selecting from the required table that houses our **model1** information
- The first row with **id = 0** specifies **Model Type**, while subsequent rows contain **Model Values**

model_table
- Columns
  - id [INTEGER, Nullable, PI]
  - model [VARCHAR (255), Nullable]

id	model
0	SEGMENT_ROWS
1	3:0.4:UNIFORM:NA
2	3:0.3:LAST_CLICK:NA
3	3:0.2:EXPONENTIAL:0.5,ROW
4	3:0.1:FIRST_CLICK:NA

Model Types: Introduction

- There are five **Attribution** Distribution Model Types
  - **SIMPLE**
  - **EVENT_REGULAR**
  - **EVENT_OPTIONAL**
  - **SEGMENT_ROWS**
  - **SEGMENT_SECONDS**
- The Model Type chosen determines **which rows qualify** as attributable events towards the defined 'conversion' event
- No matter the 'Type' chosen, the model will always scrutinize rows according to the **WindowSize** argument
- **Attribution** will be apportioned according to the parameters defined in the Model's arguments

Model Types: Introduction (cont.)

Model Type	Specification	Description
SIMPLE	*MODEL:PARAMETERS*	Distribution model for all events
EVENT_REGULAR	EVENT:WEIGHT:MODEL:PARAMETERS	Distribution model for a regular event. EVENT cannot be a conversion, excluded, or optional event. Sum of WEIGHT values must be 1.0
EVENT_OPTIONAL	EVENT:WEIGHT:MODEL:PARAMETERS	Same as EVENT_REGULAR, except for an optional event
SEGMENT_ROWS	Ki:WEIGHT:MODEL:PARAMETERS	Sum of K values. Must be value K specified by 'rows:K' in **WindowSize** argument. Function considers rows from most to least recent
SEGMENT_SECONDS	Ki:WEIGHT:MODEL:PARAMETERS	Distribution model by time in seconds. Sum of Kivalues must be value K specified by 'seconds:K' in WindowSize argument. Function considers rows from most to least recent

Lab 6a: Create Conversion Event Table and Model1 Table

teradata.


```
CREATE TABLE conversion_event_table
(conversion_events varchar (255));

INSERT INTO conversion_event_table
(conversion_events) values ('buy');

SELECT * FROM
conversion_event_table;
```

```
CREATE TABLE model_table
(id integer, model varchar (255));

insert into model_table (id, model)
values (0, 'SIMPLE');

insert into model_table (id, model)
values (1, 'UNIFORM:NA');
```

- Here, we are building, populating, and selecting from the required table that houses our conversion event of **buy**
- If we had multiple **conversion events**, they would each occupy their own row

- Here, we are building, populating, and selecting from the required table that houses our **model1** information
- The first row with **id** = **0** specifies **Model Type**, while subsequent row(s) contain **Model Values**

Lab 6b: WindowSize(Rows)

teradata.

WindowSize argument allows us to define *eligibility for attribution*

- **'rows:K' considers maximum number of rows want to attribute**
- **'seconds:K' considers maximum time difference current impact event and earliest cause event to be attributed (how many seconds to attribute)**
- **'rows:K&seconds:K' considers both constraints and chooses the most strict**

Query: Score 8 rows Uniformly prior to Buy

```
SELECT * FROM Attribution
(ON attrib7
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('rows:8')
) AS dt order by id, ts;
```

	id	event	ts	attribution	time_to_conve
1	1	BannerAd	2001-09-27 23:00:01.000000	0	
2	1	BannerAd	2001-09-27 23:00:03.000000	0	
3	1	PaidSearch	2001-09-27 23:00:05.000000	0.125	-15
4	1	InStorePromo	2001-09-27 23:00:07.000000	0.125	-13
5	1	TV	2001-09-27 23:00:09.000000	0.125	-11
6	1	TV	2001-09-27 23:00:11.000000	0.125	-9
7	1	PrintAd	2001-09-27 23:00:13.000000	0.125	-7
8	1	Email	2001-09-27 23:00:15.000000	0.125	-5
9	1	PrintAd	2001-09-27 23:00:17.000000	0.125	-3
10	1	PrintAd	2001-09-27 23:00:19.000000	0.125	-1
11	1	Buy	2001-09-27 23:00:20.000000		

8 rows prior to Conversion row

Find 8 rows prior to Buy and apportion out weight Uniformly

Lab 6bc: WindowSize(Seconds)

teradata.

Query: Score rows 17 seconds Uniformly prior to Conversion = Buy

```
SELECT * FROM Attribution
(ON attrib7
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1
DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('seconds:17')
) AS dt order by id, ts;
```

Buy occurred at 23:00:20.

So only looking for rows where timestamp between 23:00:03 and 23:00:20

Input – attrib7

	id	event	ts
1	1	BannerAd	2001-09-27 23:00:01.000000
2	1	BannerAd	2001-09-27 23:00:03.000000
3	1	PaidSearch	2001-09-27 23:00:05.000000
4	1	InStorePromo	2001-09-27 23:00:07.000000
5	1	TV	2001-09-27 23:00:09.000000
6	1	TV	2001-09-27 23:00:11.000000
7	1	PrintAd	2001-09-27 23:00:13.000000
8	1	Email	2001-09-27 23:00:15.000000
9	1	PrintAd	2001-09-27 23:00:17.000000
10	1	PrintAd	2001-09-27 23:00:19.000000
11	1	Buy	2001-09-27 23:00:20.000000

	id	event	ts	attribution	time_to_conversion
1	1	BannerAd	2001-09-27 23:00:01.000000	0	
2	1	BannerAd	2001-09-27 23:00:03.000000	0.111111111111111	-17
3	1	PaidSearch	2001-09-27 23:00:05.000000	0.111111111111111	-15
4	1	InStorePromo	2001-09-27 23:00:07.000000	0.111111111111111	-13
5	1	TV	2001-09-27 23:00:09.000000	0.111111111111111	-11
6	1	TV	2001-09-27 23:00:11.000000	0.111111111111111	-9
7	1	PrintAd	2001-09-27 23:00:13.000000	0.111111111111111	-7
8	1	Email	2001-09-27 23:00:15.000000	0.111111111111111	-5
9	1	PrintAd	2001-09-27 23:00:17.000000	0.111111111111111	-3
10	1	PrintAd	2001-09-27 23:00:19.000000	0.111111111111111	-1
11	1	Buy	2001-09-27 23:00:20.000000		

17 seconds prior to Conversion row

teradata.

Lab 6d: WindowSize(Rows&Seconds)

Syntax

```
SELECT * FROM Attribution
(ON borre_xx
 PARTITION BY user_id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1
 USING
 EventColumn ('event')
TimestampColumn ('ts')
 WindowSize ('rows:4&seconds:5')
) AS dt
ORDER BY user_id, ts;
```

Multiple 'ConversionEvents'

conversion_events
buyonline
buyinstore

seconds:5 used here since it results in fewer rows than does **rows:4**

Output

	user_id	event	ts	attribution	time_to_conversion
1	1	a	2017-01-01 13:21:00.000000	0	
2	1	b	2017-01-01 13:21:03.000000	0	
3	1	c	2017-01-01 13:21:06.000000	0	
4	1	d	2017-01-01 13:21:09.000000	0	
5	1	e	2017-01-01 13:21:12.000000	0	
6	1	f	2017-01-01 13:21:15.000000	0	
7	1	g	2017-01-01 13:21:18.000000	0	
8	1	h	2017-01-01 13:21:21.000000	0	
9	1	i	2017-01-01 13:21:24.000000	1	-3
10	1	buyonline	2017-01-01 13:21:27.000000		
11	1	a	2017-01-02 13:21:10.000000	0	
12	1	b	2017-01-02 13:21:11.000000	0	
13	1	c	2017-01-02 13:21:12.000000	0	
14	1	d	2017-01-02 13:21:13.000000	0	
15	1	e	2017-01-02 13:21:14.000000	0	
16	1	f	2017-01-02 13:21:15.000000	0.25	-4
17	1	g	2017-01-02 13:21:16.000000	0.25	-3
18	1	h	2017-01-02 13:21:17.000000	0.25	-2
19	1	i	2017-01-02 13:21:18.000000	0.25	-1
20	1	buyinstore	2017-01-02 13:21:19.000000		

rows:4 used here since it results in fewer rows than does **seconds:5**

Model Types: Input Data

- The next pages will walk through different examples and Labs of various model types
- All examples and Labs will reference the **attrib9** table

- attrib9
  - Columns
    - id [INTEGER, Nullable, PI]
    - event [VARCHAR (64), Nullable]
    - ts [TIMESTAMP, Nullable]

	id	event	ts
1	1	StorePromo	2001-09-27 23:00:13.000000
2	1	StorePromo	2001-09-27 23:00:05.000000
3	1	StorePromo	2001-09-27 23:00:09.000000
4	1	StorePromo	2001-09-27 23:00:01.000000
5	1	StorePromo	2001-09-27 23:00:07.000000
6	1	Email	2001-09-27 23:00:15.000000
7	1	StorePromo	2001-09-27 23:00:03.000000
8	1	StorePromo	2001-09-27 23:00:11.000000
9	1	Buy	2001-09-27 23:00:20.000000
10	1	StorePromo	2001-09-27 23:00:17.000000
11	1	Email	2001-09-27 23:00:19.000000

Lab 7: Model Types: SIMPLE

Query: Score closest 4 rows Uniformly prior to Conversion = Buy

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	0.25	-7
8	1	Email	2001-09-27 23:00:15.000000	0.25	-5
9	1	StorePromo	2001-09-27 23:00:17.000000	0.25	-3
10	1	Email	2001-09-27 23:00:19.000000	0.25	-1
11	1	Buy	2001-09-27 23:00:20.000000		

The **4 rows** leading up to **Buy** receive 100% percent of **Attribution** spread **Uniformly**

Lab 8: Model Types: EVENT_REGULAR

Query: Score closest 4 rows Uniformly if Touchpoint = Email (due to 'EVENT_REGULAR') prior to Conversion = Buy

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversio
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	Email	2001-09-27 23:00:15.000000	0.5	-5
8	1	Email	2001-09-27 23:00:19.000000	0.5	-1
9	1	Buy	2001-09-27 23:00:20.000000		

Rows of event **Email** that are within **4 rows** of event **Buy** receive 100% percent of the **Attribution** spread in a **Uniform** fashion

Lab 9: Model Types: SEGMENT_ROWS

Query: Score closest 4 rows where 2 closest rows get 70% of weight Uniformly and next 2 rows get 30% of weight Uniformly prior to Conversion = Buy

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	0.15	-7
8	1	Email	2001-09-27 23:00:15.000000	0.15	-5
9	1	StorePromo	2001-09-27 23:00:17.000000	0.35	-3
10	1	Email	2001-09-27 23:00:19.000000	0.35	-1
11	1	Buy	2001-09-27 23:00:20.000000		

Out of the **four rows** leading up to **Buy**:

- The **2 closest** will receive **70%** of the **Attribution** spread **Uniformly**
- The **next 2 closest** will receive **30%** of the **Attribution** spread **Uniformly**

Lab 10: Model Types: SEGMENT_SECONDS

Query: Score rows within 12 seconds prior to Conversion = Buy.
Rows within 5 seconds get 70% weight Uniformly. Remaining rows get 30% scored Uniformly

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('SECONDS:12')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0.09999999999999999	-11
6	1	StorePromo	2001-09-27 23:00:11.000000	0.09999999999999999	-9
7	1	StorePromo	2001-09-27 23:00:13.000000	0.09999999999999999	-7
8	1	Email	2001-09-27 23:00:15.000000	0.2333333333333333	-5
9	1	StorePromo	2001-09-27 23:00:17.000000	0.2333333333333333	-3
10	1	Email	2001-09-27 23:00:19.000000	0.2333333333333333	-1
11	1	Buy	2001-09-27 23:00:20.000000		

For rows that are within **12 seconds** prior to **Buy**:

- The ones within **5 seconds** will receive **70%** of the **Attribution** spread in a **Uniform** fashion
- The ones within the next **7 seconds** will receive **30%** of the **Attribution** spread in a **Uniform** fashion

Model Values: Introduction

- There are five Model Values and parameters
  - LAST_CLICK
  - FIRST_CLICK
  - UNIFORM
  - EXPONENTIAL
  - WEIGHTED
- The Model Value and parameters chosen determines the following:
  - The rows against which attribution will be apportioned
  - The weight that qualifying, attributable rows will receive

Model Values: Introduction (cont.)

Model	Description	Parameters
'LAST_CLICK'	Conversion event is attributed entirely to most recent attributable event	'NA'
'FIRST_CLICK'	Conversion event is attributed entirely to first attributable event	'NA'
'UNIFORM'	Conversion event is attributed uniformly to preceding attributable events	'NA'
'EXPONENTIAL'	Conversion event is attributed exponentially to preceding attributable events (the more recent the event, the higher the attribution)	*'alpha,type'* where *alpha* is a decay factor in range (0, 1) and type is ROW, MILLISECOND, SECOND, MINUTE, HOUR, DAY, MONTH, or YEAR. When *alpha* is in range (0, 1), sum of series wi=(*1-alpha*)**alphai* is 1. Function uses wi as exponential weights
'WEIGHTED'	Conversion event is attributed to preceding attributable events with weights specified by PARAMETERS.SEGMENT_SECONDS (when you specify 'rows:K&seconds:K' in Window argument)	You can specify any number of weights. If there are more attributable events than weights, extra (least recent) events are assigned zero weight. If there are more weights than attributable events, then function renormalizes weights

Lab 11: Model Values: 'LAST_CLICK'

Query: Score closest row to Conversion = Buy with Weight = 1

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	0	
8	1	Email	2001-09-27 23:00:15.000000	0	
9	1	StorePromo	2001-09-27 23:00:17.000000	0	
10	1	Email	2001-09-27 23:00:19.000000	1	-1
11	1	Buy	2001-09-27 23:00:20.000000		

Of the **4 rows** leading up to **Buy**, the **last event** receives **100%** percent of the **Attribution**

Lab 12: Model Values: 'FIRST_CLICK'

Query: Find 4th row from Conversion = Buy and assign Weight = 1

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	1	-7
8	1	Email	2001-09-27 23:00:15.000000	0	
9	1	StorePromo	2001-09-27 23:00:17.000000	0	
10	1	Email	2001-09-27 23:00:19.000000	0	
11	1	Buy	2001-09-27 23:00:20.000000		

Of the 4 rows leading up to Buy, the first event receives 100% percent of the **Attribution**

teradata.

Lab 13: Model Values: 'UNIFORM'

Query: 4 rows closest to Conversion = Buy are Uniformly Weighted same

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	0.25	-7
8	1	Email	2001-09-27 23:00:15.000000	0.25	-5
9	1	StorePromo	2001-09-27 23:00:17.000000	0.25	-3
10	1	Email	2001-09-27 23:00:19.000000	0.25	-1
11	1	Buy	2001-09-27 23:00:20.000000		

The **4 rows** leading up to **Buy** receive **100%** percent of **Attribution** spread **Uniformly**

Lab 14: Model Values: 'EXPONENTIAL'

Query: Starting at row closest to Conversion = Buy, decay by 50% Weight for 4 rows

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

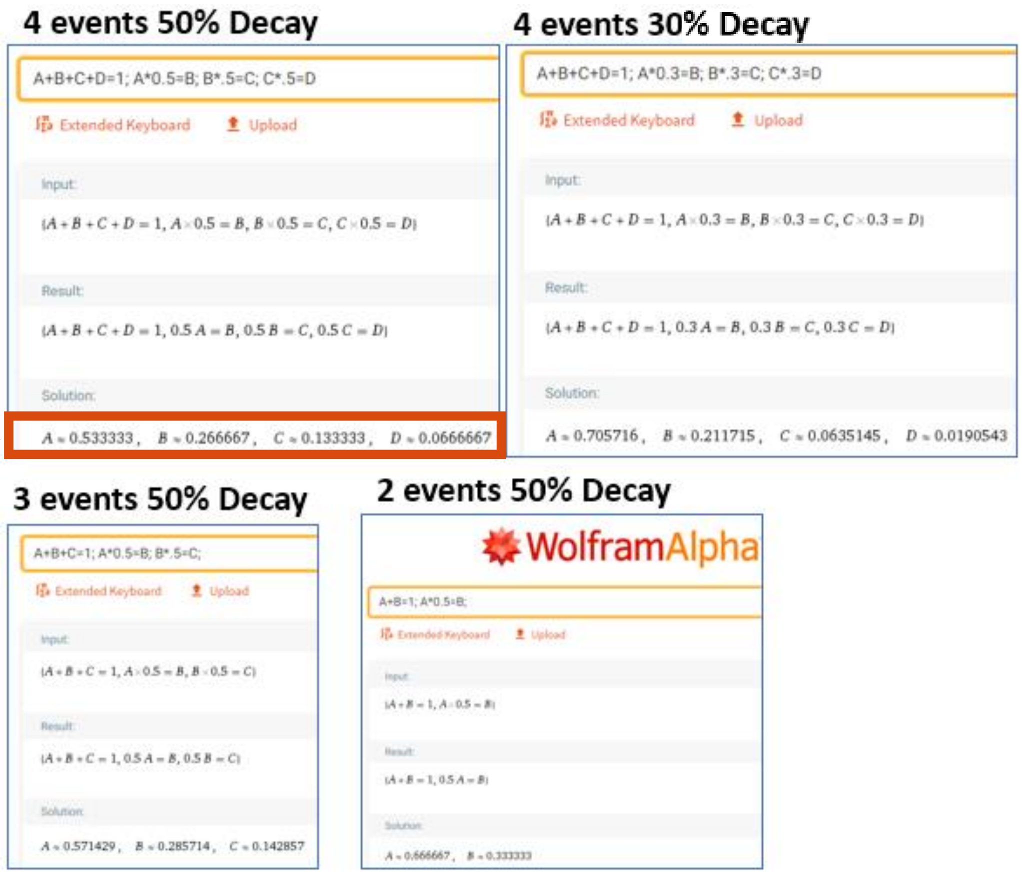
	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.0...	0	
2	1	StorePromo	2001-09-27 23:00:03.0...	0	
3	1	StorePromo	2001-09-27 23:00:05.0...	0	
4	1	StorePromo	2001-09-27 23:00:07.0...	0	
5	1	StorePromo	2001-09-27 23:00:09.0...	0	
6	1	StorePromo	2001-09-27 23:00:11.0...	0	
7	1	StorePromo	2001-09-27 23:00:13.0...	0.0666666666666667	-7
8	1	Email	2001-09-27 23:00:15.0...	0.1333333333333333	-5
9	1	StorePromo	2001-09-27 23:00:17.0...	0.2666666666666666	-3
10	1	Email	2001-09-27 23:00:19.0...	0.5333333333333333	-1
11	1	Buy	2001-09-27 23:00:20.0...		

- Of the **4 rows** leading up to **Buy**, we have defined a **50% exponential decay**, beginning with the closest row to **Buy**
- Each **previous row** will receive half the **attribution** as the one after it.
- All qualifying rows will add up to **100% attribution**

How to Calculate Exponential Weight

	id	model
1	0	SIMPLE
2	1	EXPONENTIAL:0.5,ROW

attribution
0.0666666666666667
0.133333333333333
0.266666666666666
0.533333333333333



Lab 15: Model Values: 'WEIGHTED'

Query: Weigh 4 rows individually based on Percentages defined in Model1

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib9
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:4')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	StorePromo	2001-09-27 23:00:01.000000	0	
2	1	StorePromo	2001-09-27 23:00:03.000000	0	
3	1	StorePromo	2001-09-27 23:00:05.000000	0	
4	1	StorePromo	2001-09-27 23:00:07.000000	0	
5	1	StorePromo	2001-09-27 23:00:09.000000	0	
6	1	StorePromo	2001-09-27 23:00:11.000000	0	
7	1	StorePromo	2001-09-27 23:00:13.000000	0.1	-7
8	1	Email	2001-09-27 23:00:15.000000	0.1	-5
9	1	StorePromo	2001-09-27 23:00:17.000000	0.3	-3
10	1	Email	2001-09-27 23:00:19.000000	0.5	-1
11	1	Buy	2001-09-27 23:00:20.000000		

- Of the **4 rows** leading up to **Buy**, we have **manually defined** how weights should be apportioned
- The closest row preceding **Buy** receives **50%**. The next closest rows receive **30%**, **10%**, and **10%**, respectively
- All qualifying rows will add up to **100% attribution**

What if WindowSize Exceeds Available Data?

- At times, your **WindowSize** argument (whether **ROW** or **SECONDS** or **ROW&SECONDS**) may exceed the amount of input data available by which to apportion the **Attribution**
- In these cases, your arguments will be re-calibrated so as to **score which rows are available**
- For the following labs, assume the input data to the right, and a **ConversionEvents** value of '**Conversion**'
  - For the first Conversion, there are **8 eligible rows** prior
  - For the second Conversion, there are only **2 eligible rows** prior

ANSI SQL

```
SELECT * FROM attrib1
ORDER BY id, ts;
```

	id	event	ts
1	1	Impression	2001-09-27 23:00:01.000000
2	1	Impression	2001-09-27 23:00:03.000000
3	1	Impression	2001-09-27 23:00:05.000000
4	1	Impression	2001-09-27 23:00:07.000000
5	1	Impression	2001-09-27 23:00:09.000000
6	1	Impression	2001-09-27 23:00:11.000000
7	1	Impression	2001-09-27 23:00:13.000000
8	1	Email	2001-09-27 23:00:15.000000
9	1	Conversion	2001-09-27 23:00:16.000000
10	1	Impression	2001-09-27 23:00:17.000000
11	1	Impression	2001-09-27 23:00:19.000000
12	1	Conversion	2001-09-27 23:00:20.000000

Lab 16: WindowSize Exceeds Available Data

Syntax

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib1
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:5')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	Impression	2001-09-27 23:00:01.0...	0	
2	1	Impression	2001-09-27 23:00:03.0...	0	
3	1	Impression	2001-09-27 23:00:05.0...	0	
4	1	Impression	2001-09-27 23:00:07.0...	0.2	-9
5	1	Impression	2001-09-27 23:00:09.0...	0.2	-7
6	1	Impression	2001-09-27 23:00:11.0...	0.2	-5
7	1	Impression	2001-09-27 23:00:13.0...	0.2	-3
8	1	Email	2001-09-27 23:00:15.0...	0.2	-1
9	1	Conversion	2001-09-27 23:00:16.0...		
10	1	Impression	2001-09-27 23:00:17.0...	0.5	-3
11	1	Impression	2001-09-27 23:00:19.0...	0.5	-1
12	1	Conversion	2001-09-27 23:00:20.0...		

- We specified to evaluate **5 rows** prior to Conversion event
- The 1st Conversion (Orange) had **5 prior rows** scored uniformly
- The second Conversion (Blue) had only **2 prior rows** scored uniformly, because that is all that was available

Understanding ExcludeEvents

- We can use the optional **ExcludeEvents** argument to explicitly have our **Attribution** model not apportion any attribution to values that we specify
- The next pages will show examples of this
- All examples assume the data to the right

```
SELECT * FROM attrib7
ORDER BY id, ts;
```

	id	event	ts
1	1	BannerAd	2001-09-27 23:00:01.000000
2	1	BannerAd	2001-09-27 23:00:03.000000
3	1	PaidSearch	2001-09-27 23:00:05.000000
4	1	InStorePromo	2001-09-27 23:00:07.000000
5	1	TV	2001-09-27 23:00:09.000000
6	1	TV	2001-09-27 23:00:11.000000
7	1	PrintAd	2001-09-27 23:00:13.000000
8	1	Email	2001-09-27 23:00:15.000000
9	1	PrintAd	2001-09-27 23:00:17.000000
10	1	PrintAd	2001-09-27 23:00:19.000000
11	1	Buy	2001-09-27 23:00:20.000000

Lab 17a: Without ExcludeEvents

Syntax (without ExcludeEvents)

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib7
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1
  DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:10')
) AS dt ORDER BY id, ts;
```

Output

	id	event	ts	attribution	time_to_conversion
1	1	BannerAd	2001-09-27 23:00:01.000000	0.1	-19
2	1	BannerAd	2001-09-27 23:00:03.000000	0.1	-17
3	1	PaidSearch	2001-09-27 23:00:05.000000	0.1	-15
4	1	InStorePromo	2001-09-27 23:00:07.000000	0.1	-13
5	1	TV	2001-09-27 23:00:09.000000	0.1	-11
6	1	TV	2001-09-27 23:00:11.000000	0.1	-9
7	1	PrintAd	2001-09-27 23:00:13.000000	0.1	-7
8	1	Email	2001-09-27 23:00:15.000000	0.1	-5
9	1	PrintAd	2001-09-27 23:00:17.000000	0.1	-3
10	1	PrintAd	2001-09-27 23:00:19.000000	0.1	-1
11	1	Buy	2001-09-27 23:00:20.000000		

Note '**TV**' and '**PrintAd**' are included in the Output

Lab 17b: With ExcludeEvents

Input (with ExcludeEvents)

```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.attrib7
 PARTITION BY id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON excluding_event_table AS
ExcludedEventTable DIMENSION
 ON model_table AS model1 DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('ROWS:10')
) AS dt ORDER BY id, ts;
```

excluding_events
TV
PrintAd

Output

	id	event	ts	attribution	time_to_conversion
1	1	BannerAd	2001-09-27 23:00:01.000000	0.2	-19
2	1	BannerAd	2001-09-27 23:00:03.000000	0.2	-17
3	1	PaidSearch	2001-09-27 23:00:05.000000	0.2	-15
4	1	InStorePromo	2001-09-27 23:00:07.000000	0.2	-13
5	1	Email	2001-09-27 23:00:15.000000	0.2	-5
6	1	Buy	2001-09-27 23:00:20.000000		

Note that we can specify Multiple Events to exclude by separating them with commas

Note '**TV**' and '**PrintAd**' are excluded in the Output

Multiple Model Values Together

- There may be times you would like to combine Multiple Model Values together (Last_click, First_click, Uniform, etc.)
- Assume the input data to the right. Note the events occur at one-second intervals
- We will attempt to use a **SEGMENT_ROWS** Model Type and apportion out **attribution** to the 12 rows that precede the conversion event = 'buy'
- The 12 qualifying rows will receive the weights that we specify, and they will be scored according to our defined Multiple Model Value arguments

Input

```
SELECT * FROM borre_y
ORDER BY user_id, ts;
```

	user_id	event	ts
1	1	a	2017-01-01 13:21:01.000000
2	1	a	2017-01-01 13:21:02.000000
3	1	a	2017-01-01 13:21:03.000000
4	1	a	2017-01-01 13:21:04.000000
5	1	a	2017-01-01 13:21:05.000000
6	1	a	2017-01-01 13:21:06.000000
7	1	a	2017-01-01 13:21:07.000000
8	1	a	2017-01-01 13:21:08.000000
9	1	a	2017-01-01 13:21:09.000000
10	1	a	2017-01-01 13:21:10.000000
11	1	a	2017-01-01 13:21:11.000000
12	1	a	2017-01-01 13:21:12.000000
13	1	a	2017-01-01 13:21:13.000000
14	1	a	2017-01-01 13:21:14.000000
15	1	a	2017-01-01 13:21:15.000000
16	1	buy	2017-01-01 13:21:16.000000

Lab 18a: Multiple-Input Model: Create Model_Table

teradata.

Syntax

```
CREATE MULTISET TABLE model_table
(id integer, model varchar (255));

insert into model_table (id, model)
values (0, 'SEGMENT_ROWS');

insert into model_table (id, model)
values (1, '3:0.4:UNIFORM:NA');

insert into model_table (id, model)
values (2, '3:0.3:LAST_CLICK:NA');

insert into model_table (id, model)
values (3, '3:0.2:EXPONENTIAL:0.5,ROW');

insert into model_table (id, model)
values (4, '3:0.1:FIRST_CLICK:NA');

SELECT * FROM model_table ORDER BY id;
```

- Here, we are building, populating, and selecting from the required table that houses our **model1** information
- The first row with **id = 0** specifies **Model Type**, while subsequent rows contain **Model Values**

model_table
Columns
id [INTEGER, Nullable, PI]
model [VARCHAR (255), Nullable]

id	model
0	SEGMENT_ROWS
1	3:0.4:UNIFORM:NA
2	3:0.3:LAST_CLICK:NA
3	3:0.2:EXPONENTIAL:0.5,ROW
4	3:0.1:FIRST_CLICK:NA

Lab 18b: Multiple Model Values


```
SELECT * FROM Attribution
(ON TRNG_TDU_TD01.borre_y
 PARTITION BY user_id
 ORDER BY ts
 ON conversion_event_table AS
conversion DIMENSION
 ON model_table AS model1
DIMENSION
 USING
 EventColumn ('event')
 TimestampColumn ('ts')
 WindowSize ('rows:12')
) AS dt ORDER BY user_id, ts;
```

- **ConversionEvents ('buy'):** The conversion event = 'buy'
- **WindowSize ('rows:12'):** We will scrutinize the 12 rows leading up to the buy event
- We will use a model type of SEGMENT_ROWS. This type of model will apportion out attribution based on how many rows prior to the buy event the row is
- Each model value is preceded by the number 3. The total value the 3s added up must equal our **WindowSize** of 12
- We will score the 3 closest rows prior to buy in a UNIFORM fashion. These 3 rows will receive 40% of the total score
- Of the next 3 closest rows, only the LAST_CLICK will be scored. It will receive 30% of the total score
- Of the next 3 closest rows, will score them using EXPONENTIAL of 50%. These rows will receive 20% of total score
- Of the next 3 closest rows, only the FIRST_CLICK will be scored. It will receive 10% of the total score
- The total score must add up to 100% (0.4 + 0.3 + 0.2 + 0.1)

Lab 18c: Multiple Models Values

(1 Model Type = 'SEGMENT_ROWS' and 4 Model Values)

teradata.

Below is the answer-set from our query with **buy** as the **conversion event**. Note that rows 1, 2, and 3 are not scored, as they did not meet our **WindowSize** argument of 12 rows

	user_id	event	ts	attribution	time_to_conversion
1	1	a	2017-01-01 13:21:01.000000	0	
2	1	a	2017-01-01 13:21:02.000000	0	
3	1	a	2017-01-01 13:21:03.000000	0	
4	1	a	2017-01-01 13:21:04.000000	0.10000000000000002	-12
5	1	a	2017-01-01 13:21:05.000000	0	
6	1	a	2017-01-01 13:21:06.000000	0	
7	1	a	2017-01-01 13:21:07.000000	0.028571428571428574	-9
8	1	a	2017-01-01 13:21:08.000000	0.05714285714285715	-8
9	1	a	2017-01-01 13:21:09.000000	0.1142857142857143	-7
10	1	a	2017-01-01 13:21:10.000000	0	
11	1	a	2017-01-01 13:21:11.000000	0	
12	1	a	2017-01-01 13:21:12.000000	0.30000000000000004	-4
13	1	a	2017-01-01 13:21:13.000000	0.13333333333333336	-3
14	1	a	2017-01-01 13:21:14.000000	0.13333333333333336	-2
15	1	a	2017-01-01 13:21:15.000000	0.13333333333333336	-1
16	1	buy	2017-01-01 13:21:16.000000		

```
…WindowSize ('rows:12')
Model1 ('SEGMENT_ROWS',
'3:0.4:UNIFORM:NA',
'3:0.3:LAST_CLICK:NA',
'3:0.2:EXPONENTIAL:0.5,ROW',
'3:0.1:FIRST_CLICK:NA')…
```

```
'3:0.1:FIRST_CLICK:NA')
Receives 10% of total score
```

```
'3:0.2:EXPONENTIAL:0.5,ROW',
Receives 20% of total score
```

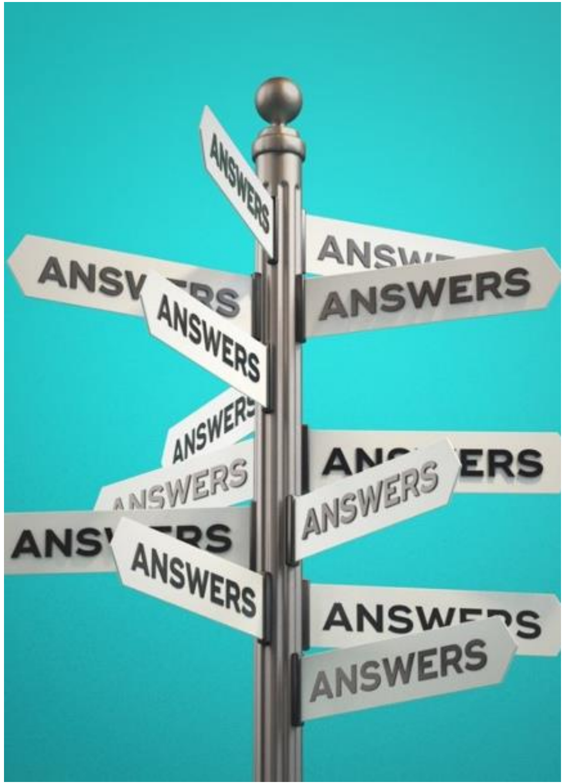
```
'3:0.3:LAST_CLICK:NA',
Receives 30% of total score
```

```
'3:0.4:UNIFORM:NA',
Receives 40% of total score
```

teradata.

Current Topic – Attribution Review

- Sessionize
  - Background Information (Description, Use Cases, Workflow, Syntax, Required Arguments, Optional Arguments, Input Table Schema, Output Table Schema)
  - Labs
  - Review
- **Attribution**
  - Background Information (Description and Use Cases)
  - Multiple-Input Models (Workflow, Syntax, Required Arguments, Optional Arguments, Input Table, Schema, Output Table Schema, Labs)
  - **Review**



Attribution Summary

Now that we have learned how to run the **Attribution** function and understand the mechanics of its output, below is one possible real-world usage of the function for you to consider

- If we have a table with all confirmed customer contacts and events, we might run a simple **Attribution** argument, apportioning attribution towards a conversion event of 'purchase' in a uniform fashion for some designated amount of time prior to the purchases. Included in this table might be events such as 'direct-mail contact', 'email contact', 'website visits', 'YouTube commercial viewed', etc. for each customer, all time-stamped. After running **Attribution**, we may find that direct-mail rows receive much more attribution towards the purchase events than all other events combined
  - Given this, we may want to re-run our **Attribution** function against the data, making sure to give a much heavier weight to rows with a value of 'direct-mail contact'
  - Given this, we may want to increase next year's budget for direct-mail, and lighten our budget on other promotional activities—which didn't seem to contribute so much towards purchase events

Summary

In this module, you learned how to:

- Describe what the Sessionize and Attribution functions do
- Describe typical use cases for Sessionize and Attribution
- Write Sessionize and Attribution queries
- Interpret the output of Sessionize and Attribution queries

Thank you.