**Database User Guide**

To create our database, we have a sql script called “G1\_darksprings.sql”. Please run this script to drop everything, create everything, and insert data. This is the first step before running any of the below code.

Examples of how to use/test each object:

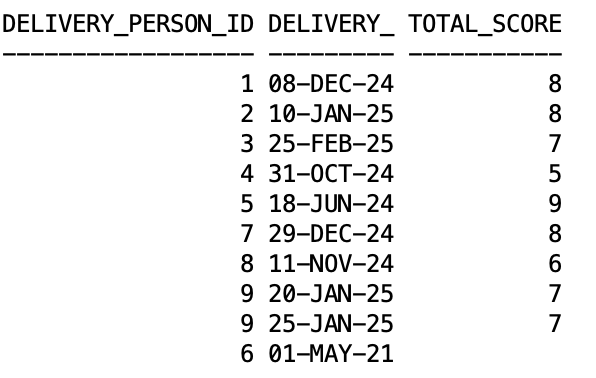
**PROCEDURES:**

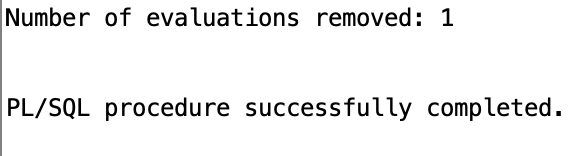
1. To test the **remove\_old\_evaluations** procedure, you can run the following code:

SET SERVEROUTPUT ON;

EXEC remove\_old\_evaluations();

This will remove all of the evaluations belonging to delivery employees that have not made a delivery in more than 3 years.





Note that there are no scores for deliveries that have a delivery date before 3 years. Run the query to check  
  
SELECT dp.delivery\_person\_id, d.delivery\_date, e.total\_score

FROM eval e right join del\_per dp on dp.delivery\_person\_id = e.delivery\_person\_id

JOIN dlvry d on d.delivery\_person\_id = dp.delivery\_person\_id.

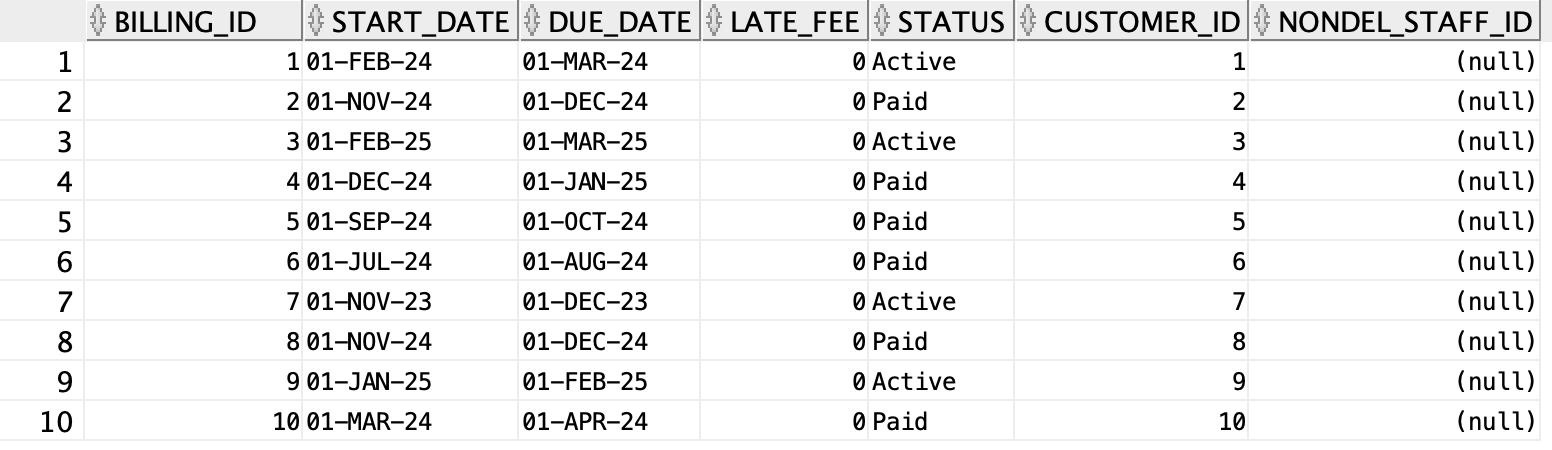
Additionally, if you look in the evaluation table you will find that the evaluation pertaining to delivery person 6 is now gone as delivery person 6 was one who had not made a delivery in more than 3 years.

1. To test the Update\_late\_Fees procedure, you can run the following code:

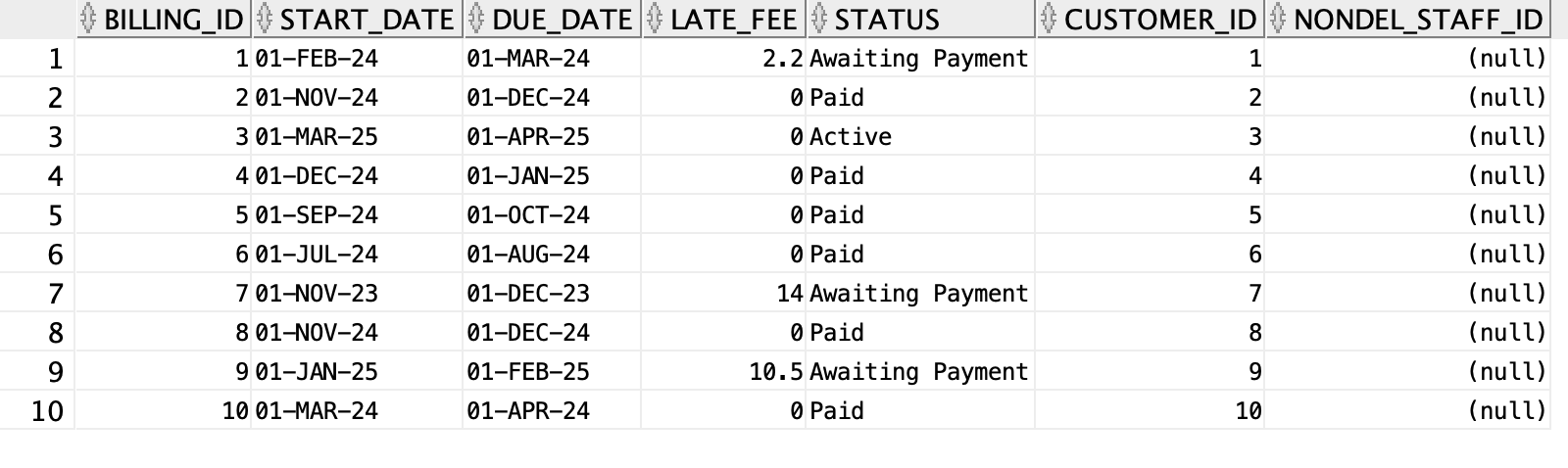
SET SERVEROUTPUT ON;

EXEC Update\_late\_Fees();

For each customer who has not paid their billing by the due date, this procedure will add a late fee following the equation: Late Fee = Total Billing Cost \* 10% \* Number of Months Late.



After running the procedure, we get the following table. To see this output, query the bills table after running the procedure.



**FUNCTIONS:**

1. To test the get\_delivery\_person\_stats function, you can run a version of the following code:

VAR performance\_review\_cursor REFCURSOR;

BEGIN

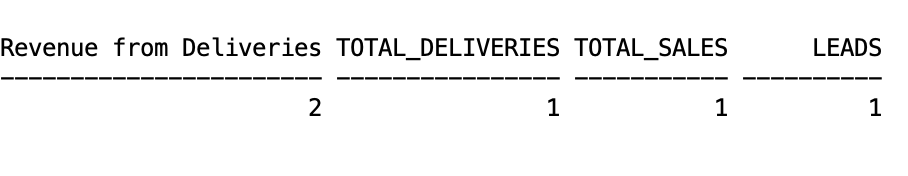
:performance\_review\_cursor := get\_delivery\_person\_stats(1, 12, 2024);

END;

/

PRINT performance\_review\_cursor;

This function will return a table that shows, for delivery person 1 in December of 2024, their number of deliveries, sales, total revenue generated from those sales, and leads recorded .



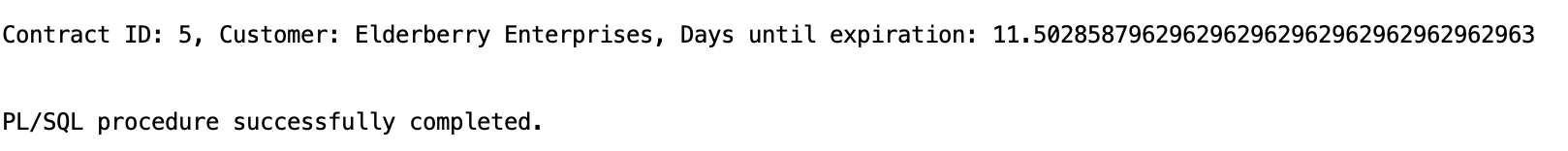
**PACKAGE 1 (Contract Actions):**

1. To run the **find\_expiring\_contract** procedure within the contract actions package, you can run the following code:

SET SERVEROUTPUT ON;

EXEC contract\_actions.find\_expiring\_contracts;

This procedure will show a list of all contracts that are set to expire within the next 30 days.



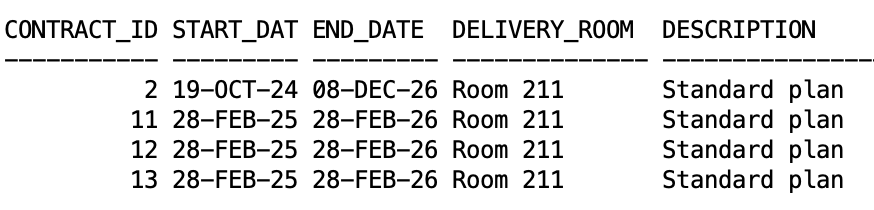
1. To run the **renew\_contract** procedure within the contract actions package, you can run the following code:

SET SERVEROUTPUT ON;

EXEC contract\_actions.renew\_contract(2); -- customer id

This code will find the active contract for the customer organization with customer\_id = 2 and renew that contract for another year retaining all of the same other features as the previous contract.

To see this output, query the CONTR table after running the procedure.



1. To run the **create\_new\_contract** procedure within the contract actions package, you can run the following code:

SET SERVEROUTPUT ON;

BEGIN

contract\_actions.create\_new\_contract(

customer\_v => 2,

delivery\_room\_v => 'Room A',

description\_v => 'Standard Plan',

estimated\_frequency\_v => 5,

initial\_charge\_v => 25.00,

estimated\_monthly\_cost\_v => 250,

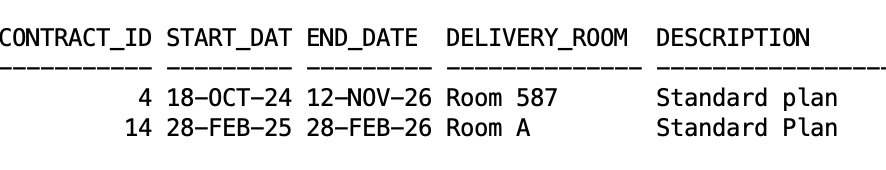
nondel\_staff\_id\_v => 4

);

END;

/

This code

****

(Run this to view SELECT \* FROM CONTR WHERE customer\_id = 4)

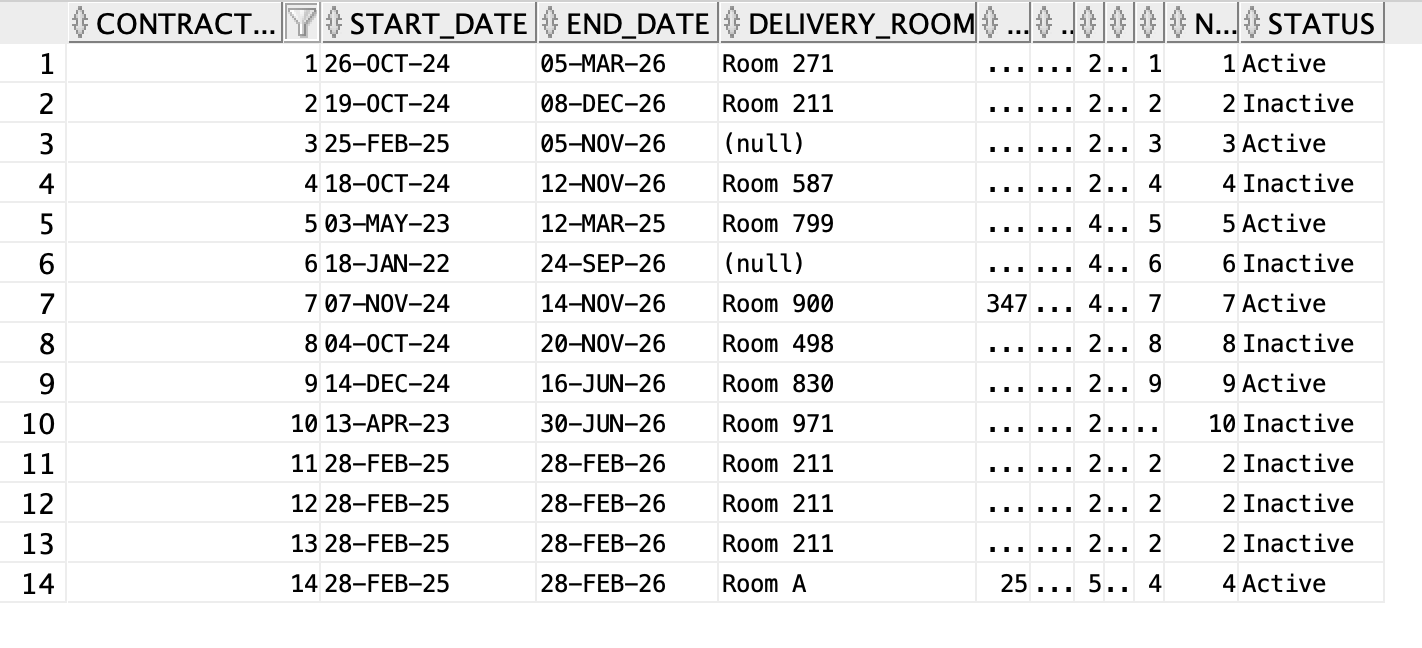
1. To run the **terminate\_contracts** procedure within the contract actions package, you can run the following code:

SET SERVEROUTPUT ON;

EXEC contract\_actions.terminate\_contracts(2); -- customer id

This code will “terminate” all active contracts for customer 2 by changing the status of the contract from “Active” to “Inactive”.

To view this output, query the CONTR table after running the package.



**PACKAGE 2 (Sales Actions):**

1. To run the **create\_new\_sale** procedure within the sales actions package, you can run the following code:

BEGIN

create\_new\_sale(

v\_billing => 2, -- Billing ID

v\_contract => 2, -- Contract ID

v\_customer => 2, -- Customer ID

v\_delivery\_person => 3, -- Delivery Person ID

v\_inventory\_1 => Cheap small water', -- Item 1

v\_inventory\_1\_quantity => 1, -- Item 1 Quantity

v\_inventory\_2 => Expensive large water', -- Item 2

v\_inventory\_2\_quantity => 1 -- Item 2 Quantity

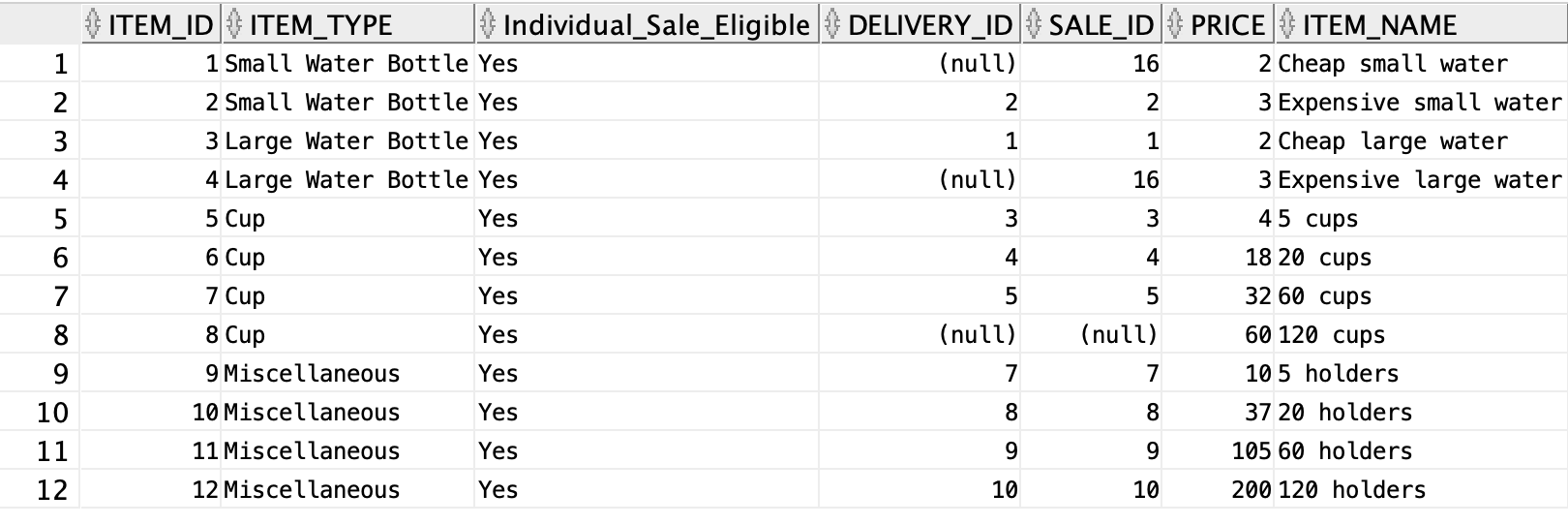
);

END;

/

This uses the sales and nonc\_inv (non-cooler inventory) tables to create a new sale. It will first create a new record in the sales table that is associated with customer 2 and their respective contract and billing. The sale is also assigned to delivery person 3 who at some point in the future will need to carry out the delivery. This particular sale asks for 1 small cheap water and 1 small cup. The procedure will then look in the inventory table and if there is 1 small cheap water bottle and 1 small cup available, it will assign those inventory items to this sale.

To view this output, query the NONC\_INV table after running the procedure.



Note: Look at Item\_ID 1 and 4

1. To run the **update\_sale** procedure within the sales actions package, you can run the following code:

BEGIN

sales\_actions.update\_sale(

v\_sale\_id => 16,

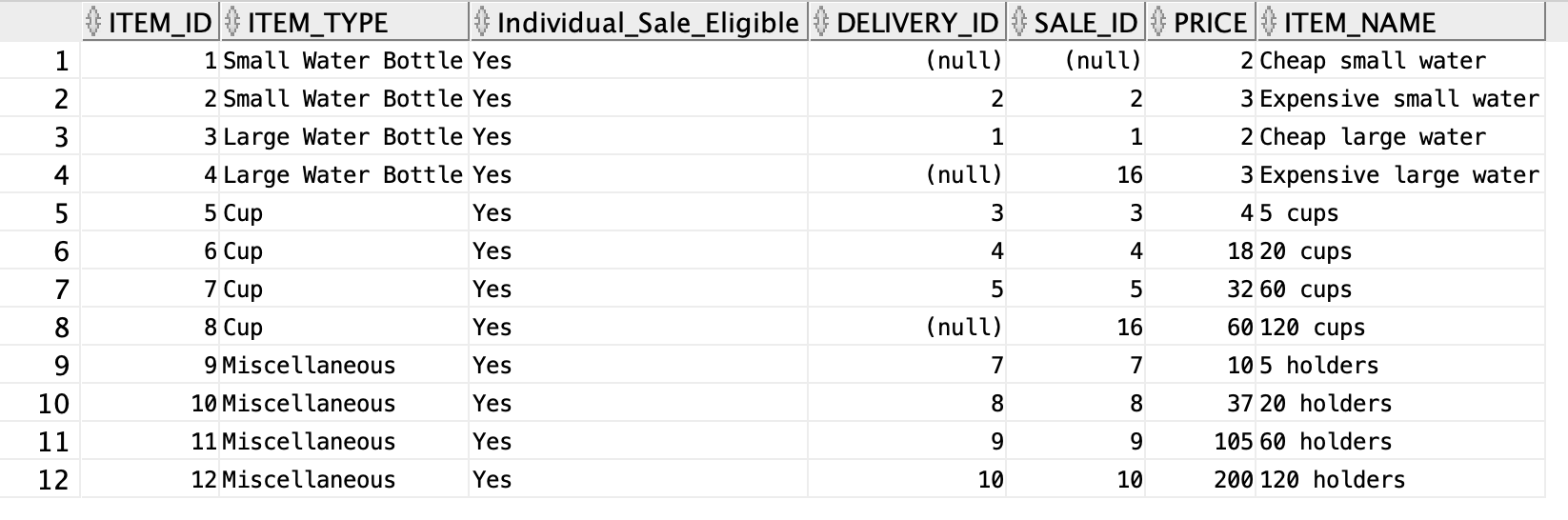
v\_old\_product => 'Cheap small water',

v\_new\_product => '120 cups'

);

END;

To view this output, query the NONC\_INV table after running the procedure.



**TRIGGERS:** Each trigger should be added automatically after running the SQL script. You can test the functionality of each trigger by following the steps below:

1. To test the **referral\_bonus** trigger you can run the following:

UPDATE Leads

SET "Approved" = 'Yes'

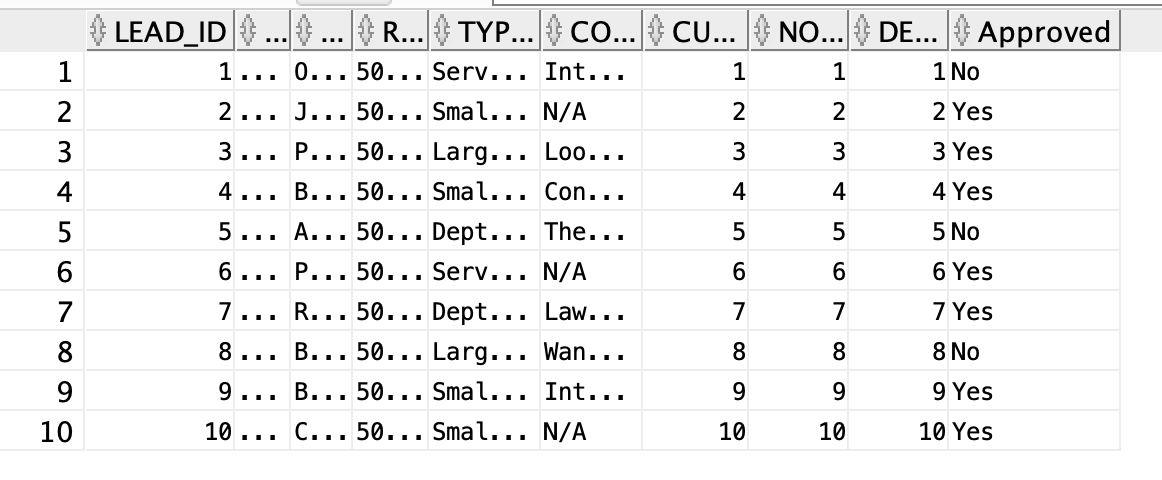
WHERE LEAD\_ID = 5;

This will change the approved status in the leads table to yes and then the trigger will automatically add $25 to the total balance of the customer who referred that particular lead.

This is the original table.

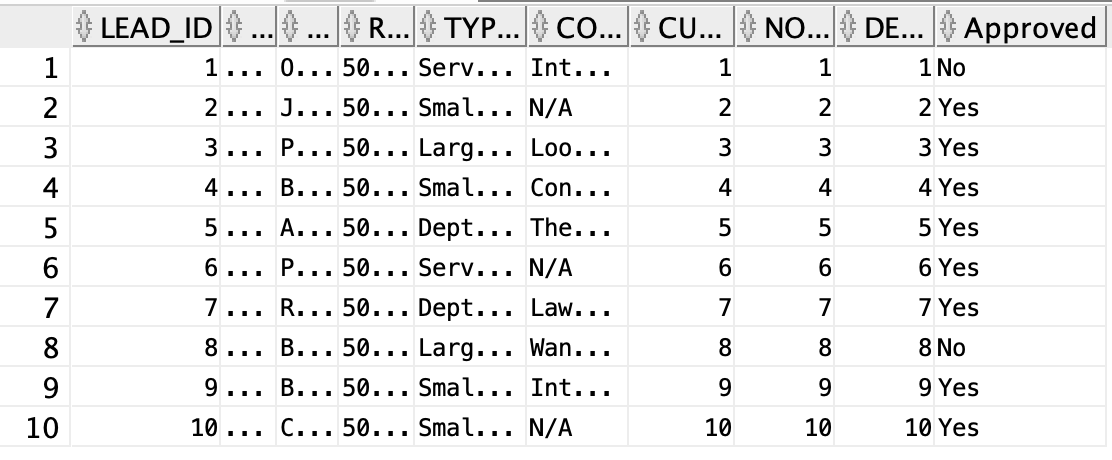
To view this output, query the LEADS table.

.



This is the changed table. Notice lead\_id = 5

To view this output, query the LEADS table after running the trigger.



1. To test the **collect\_deposit** trigger you can run the following:

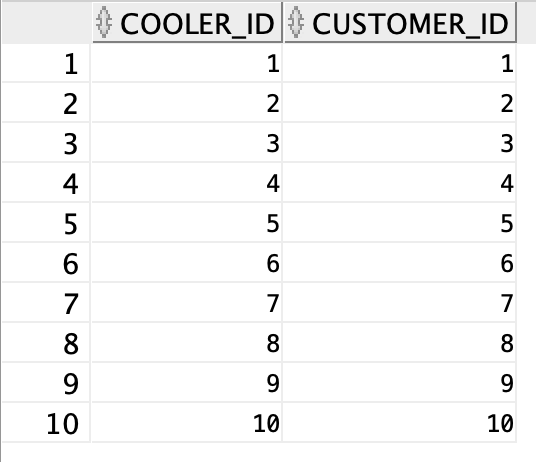
INSERT INTO cool\_rent (COOLER\_ID, CUSTOMER\_ID)

VALUES (9, 4);

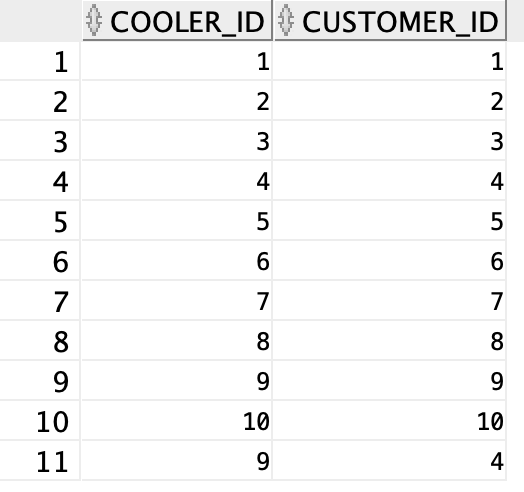
This will add a new row to the cooler rental table. The trigger then will automatically add $25 to the total balance of customer 4 which represents the total amount of money that customer 4 owes to Dark Springs.

This is the original table.

To view this output, query the COOL\_RENT table.



This is the updated table.



1. To test the **return\_deposit** trigger you can run the following:

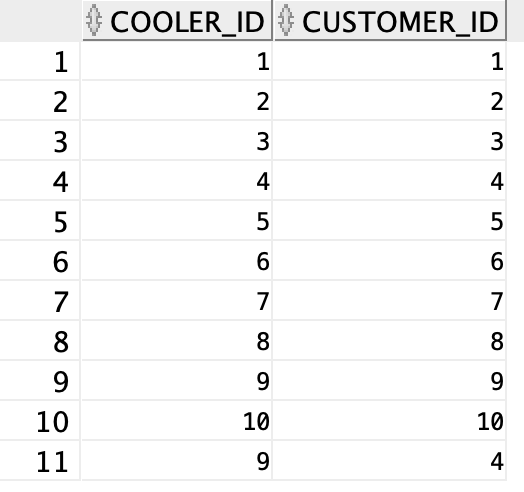
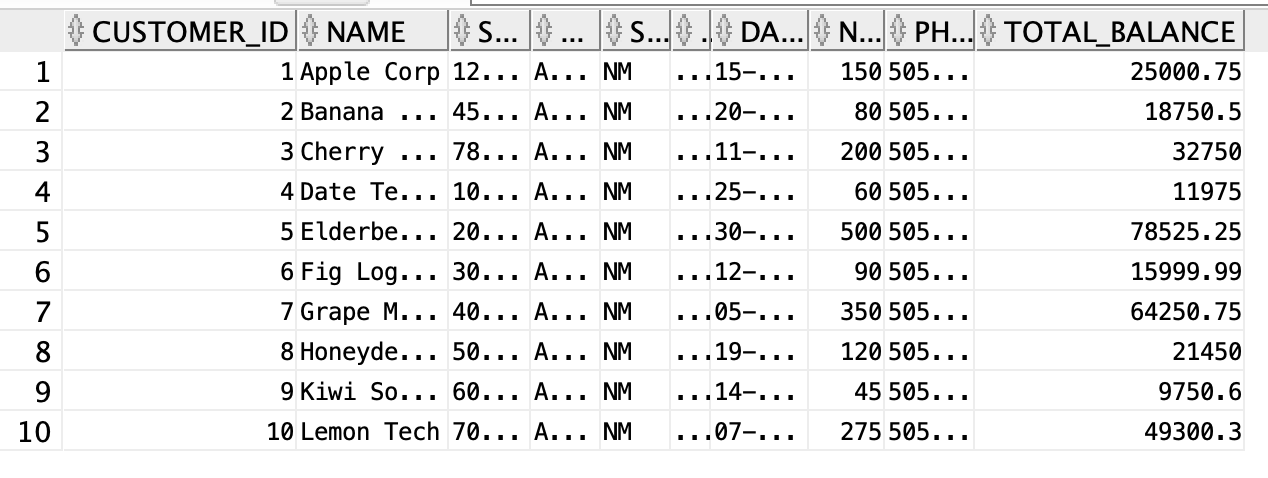
DELETE FROM COOL\_RENT

WHERE cooler\_id = 9 AND customer\_id = 4;

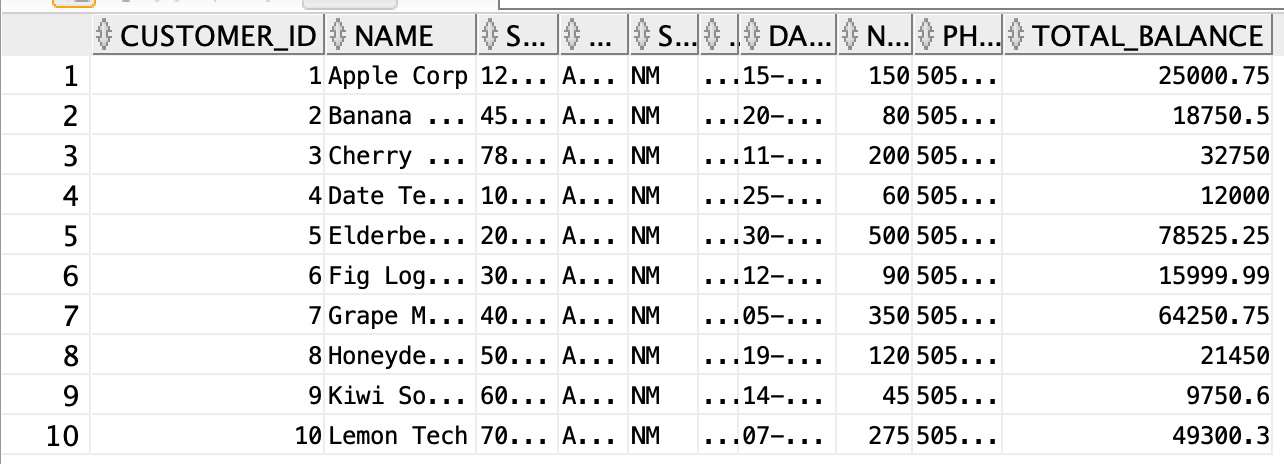
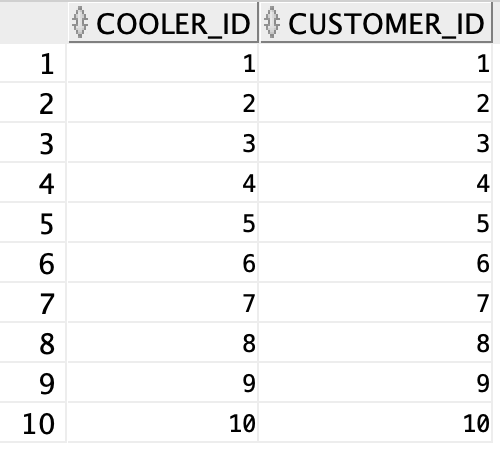
This will delete the row within the cool\_rent table that has the cooler id of 9 and the customer id of 4. The trigger then will automatically subtract $25 from customer 4’s total balance.

To view the following, query the COOL\_RENT table.

Original tables

After the update…



**SCHEDULED JOB:** The SQL script should automatically set the job. By default, it is scheduled to calculate a customer’s billing and update their balance on the first day of each month. However, you can manually change the frequency of the job/activity by using some version of the code below.

BEGIN

DBMS\_SCHEDULER.SET\_ATTRIBUTE(

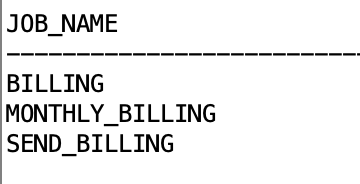
name => 'send\_billing',

attribute => 'repeat\_interval',

value => 'FREQ=WEEKLY; BYDAY=SUN; BYHOUR=12; BYMINUTE=0; BYSECOND=0'

);

END;



(Run the following query to get this

SELECT JOB\_NAME

FROM USER\_SCHEDULER\_JOBS;)

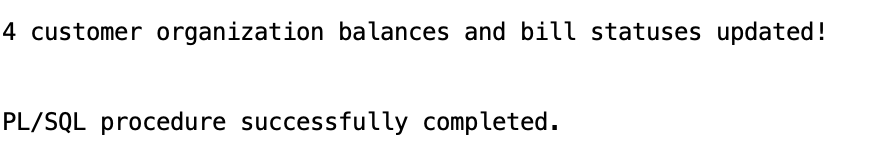
**PROCEDURE IN THE SCHEDULED JOB:**

Can be run using the following code:

SET SERVEROUTPUT ON;

EXEC Calc\_Send\_Billing();

This code chunk will calculate the sales associated with all active bills and then send that bill to the customer. To send the bill, the procedure will then update the bill in the cust\_org table and then update the status of the bill from “Active” to “Awaiting Payment”.

****

To view the updates, query the CUST\_ORG table and the BILLS table.

**ROLES:**

Run the roles.sql file. This will create the roles. Then run the following command to view them.

SELECT \* FROM USER\_ROLE\_PRIVS;To check the privileges of each role, run  
  
SELECT \*

FROM ROLE\_TAB\_PRIVS

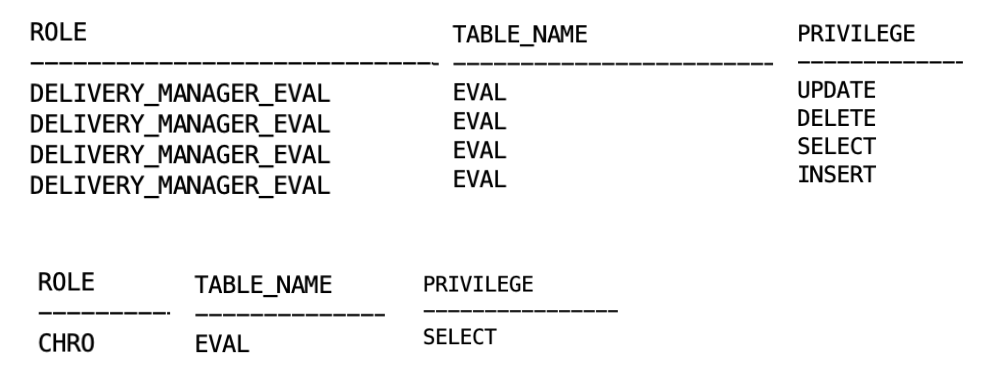
WHERE ROLE = 'DELIVERY\_MANAGER\_EVAL';

For role 2:

SELECT \*

FROM ROLE\_TAB\_PRIVS

WHERE ROLE = 'CHRO';

****

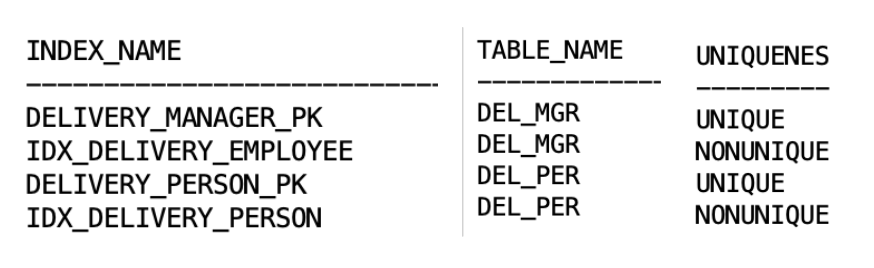
**ALTERNATE INDEXES:**

Run the alt\_index.sql file. Then run this query to view the indexes

SELECT INDEX\_NAME, TABLE\_NAME, UNIQUENESS

FROM USER\_INDEXES

WHERE TABLE\_NAME IN ('DEL\_MGR', 'DEL\_PER');

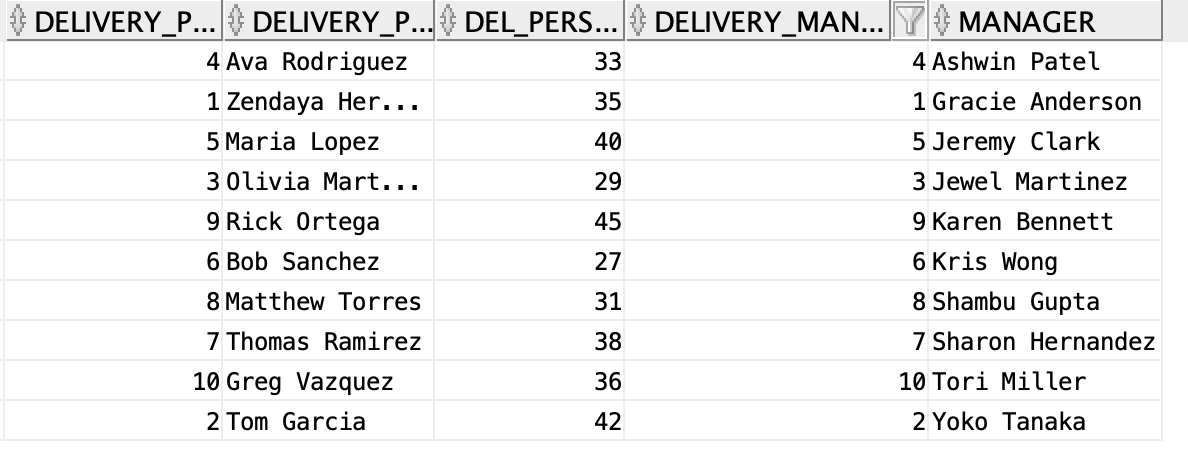
****

**DE-NORMALIZATION:**

Run denorm.sql to create the view. Then run the following query

SELECT \*

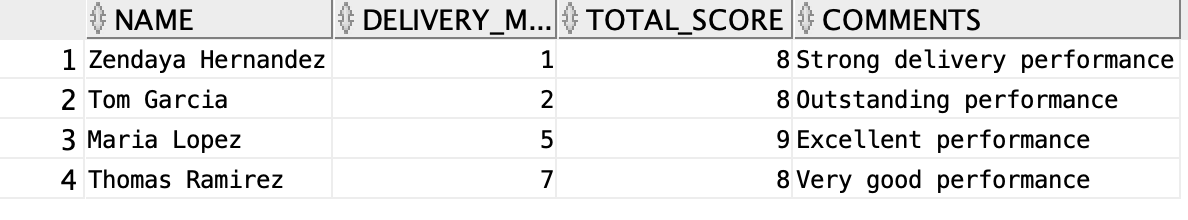
FROM DEL\_PER\_MGR



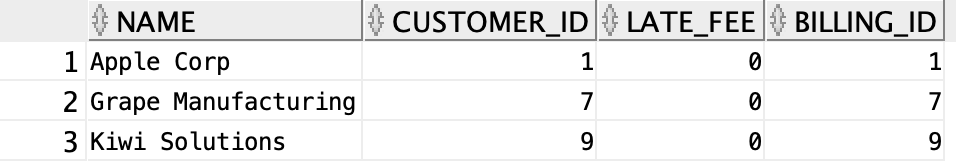
**REPORT:**

In the SQL Developer, click on the Reports Tab > User Defined Reports. Then view each of the 2 reports.

**High performing delivery persons**



**Awaiting Bill Status:**

****

SCENARIOS

**Scenario 1: Delivery Person Person Performance Review**

The business issue is that the delivery manager is having a hard time telling which delivery personnel are the most productive for the company. If the delivery manager would like to understand how the delivery people are performing, they can look at their revenue generation, delivery count, sales, and lead metrics for a specific month to understand quantitatively the performance of that employee.

The following code can be used to calculate the delivery person statistics for delivery person 9 in January of 2025:

VAR performance\_review\_cursor REFCURSOR;

BEGIN

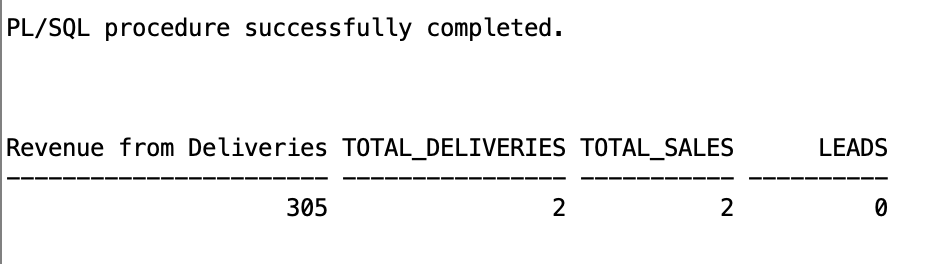
:performance\_review\_cursor := get\_delivery\_person\_stats(9, 1, 2025);

END;

/

PRINT performance\_review\_cursor;

After completing the procedure, the output will look like this. Delivery person 9 brought in $305 in revenue from deliveries, made 2 deliveries, and made 2 sales. This person did not bring in leads.



**Scenario 2: Expiring Contract Renewal**

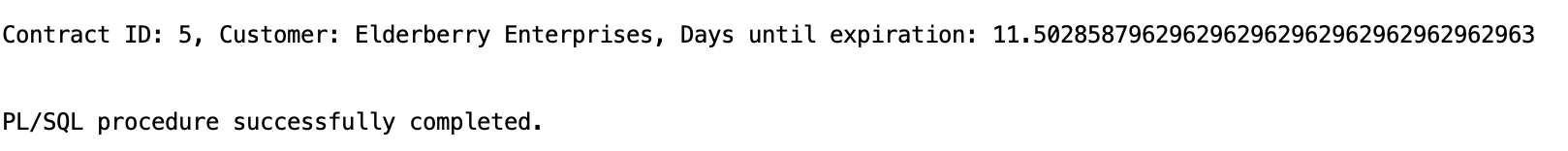
The business issue is that Dark Springs has a customer that is going to have interrupted service if the contract isn’t renewed in the system. This would be bad for business and could potentially lead to the customer cancelling their contract, so the contract\_action package finds customer’s contracts nearing expiration and renews the contract to help ensure uninterrupted service.

The following code can be used to run the find expiring contracts procedure from the contract\_action package:

SET SERVEROUTPUT ON;

EXEC contract\_actions.find\_expiring\_contracts;

The following screenshot shows the output of the above code. We can see the contract with Elderberry Enterprises is expiring in 11 days.



Then, the following code can be used to run the contract renewal procedure from the contract\_action package:

SET SERVEROUTPUT ON;

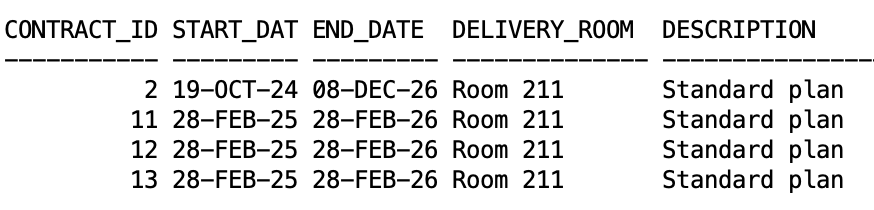
BEGIN

contract\_actions.renew\_contract(2); -- customer id

END;

/

Running the above code produces the following output. Query the CONTR table to see the change.



**Scenario 3: Referral Bonus Automation**

The business issue is that Dark Springs needs to make sure that referral incentives are applied consistently and accurately when a customer successfully refers a new lead. By automating the deduction of a referral bonus from the customer's balance when a lead is approved, the trigger helps reduce manual errors and administrative overhead while encouraging customers to bring in new business.

The following code can be used to run the referral\_bonus trigger:

UPDATE Leads

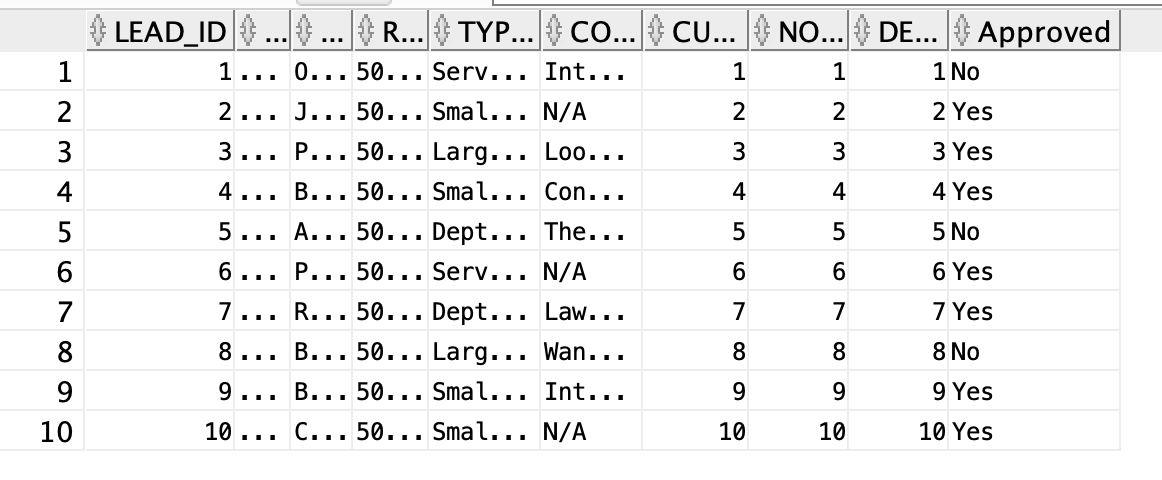
SET "Approved" = 'Yes'

WHERE LEAD\_ID = 5;

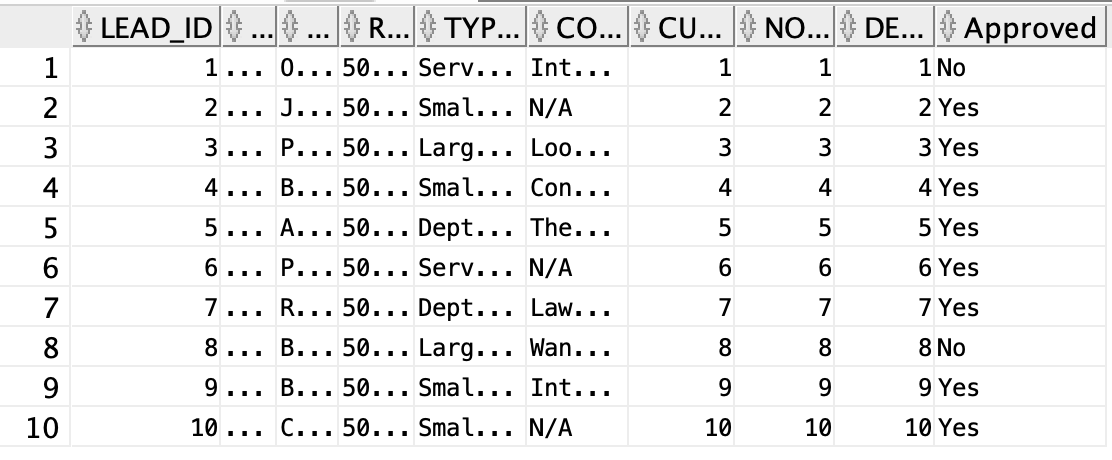
This will change the approved status in the leads table to yes and then the trigger will automatically add $25 to the total balance of the customer who referred that particular lead.

This is the original table. Query the LEADS table to view the output.

.



This is the changed table. Notice lead\_id = 5



**Scenario 4: Late Fee Calculation**

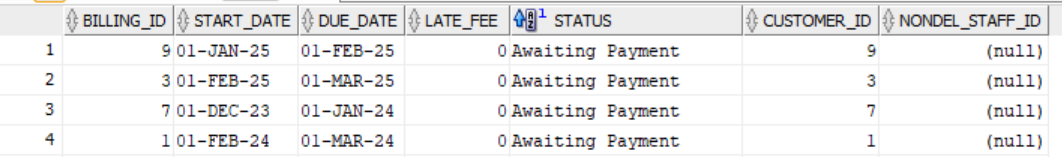
The business issue is that late payments are significantly impacting cash flow at Dark Springs and this is leading to disputes and administrative inefficiencies. The Update\_late\_Fees procedure automatically calculates and updates each overdue bill by determining the applicable late fee based on the outstanding amount and the duration of the delay. This automation ensures that late fees are applied consistently, encouraging timely payments and helping maintain a steady cash flow for the business.

The following code can be used to run the Update\_late\_Fees procedure:

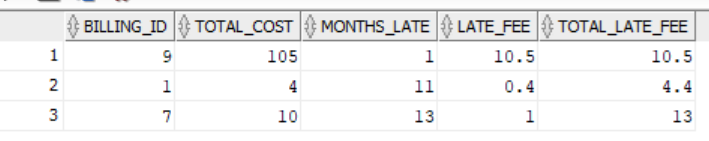
SET SERVEROUTPUT ON;

EXEC Update\_late\_Fees();

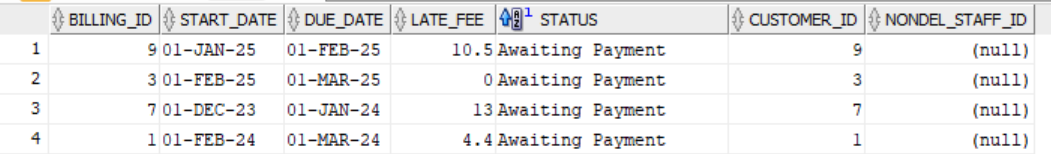
Consider the following example, in our database there are 4 bills that have not yet been paid. Three of these bills are past their due date and thus are subject to a late fee. To calculate and apply the late fee, you can run the procedure given by the code above.



The procedure will look at the the all of the sales associated with this bill and calculate the late fee as follows: Late Fee = Total Billing Cost \* 10% \* Number of Months Late. The below SQL output shows an example of how the late fee is calculated.[[1]](#footnote-0)



The procedure then updates the late fee in the bills table as shown below:



**Scenario 5: Cluttered Evaluations**

The business issue is that the database is becoming cluttered with evaluations that are not relevant. Old evaluation records for inactive delivery personnel can clutter performance data, making it difficult for Dark Springs to assess how the current employees are performing.

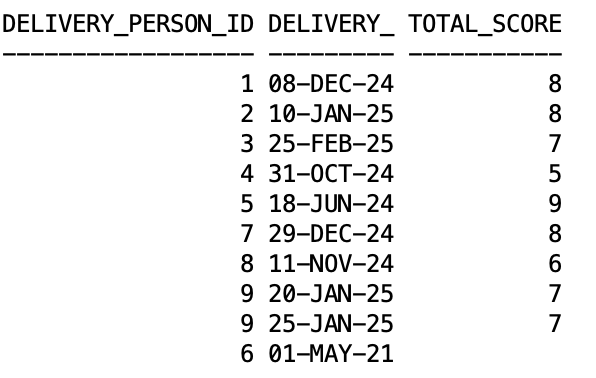
To ensure that the performance data reflects only active personnel, the delivery manager can run the remove\_old\_evaluations procedure, which automatically deletes evaluations for delivery people inactive for more than three years.

The following code can be used to run the remove\_old\_evaluations procedure:

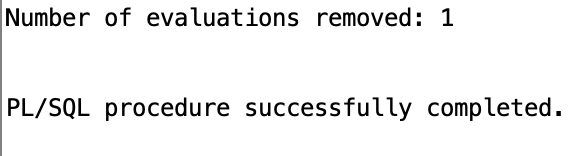
SET SERVEROUTPUT ON;

EXEC remove\_old\_evaluations();

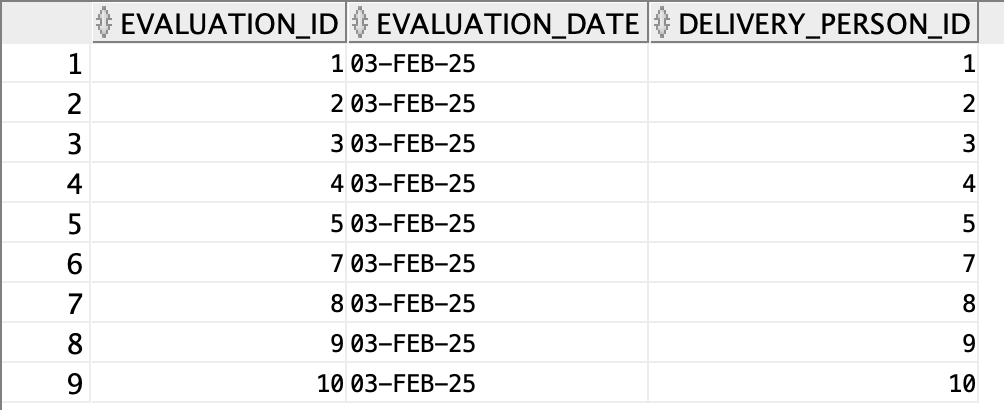
This is what the evaluation table looked like before the procedure:



After running the procedure, we can see that 1 evaluation was removed. Delivery person 6 has not done a delivery in over 3 years, so our procedure removed it.



This is what the eval table looks like after person 6 was removed.



1. This output can be obtained by running the following query:

   SELECT

   b.billing\_id,

   (COALESCE(SUM(n.price), 0)) AS total\_cost,

   GREATEST(1, TRUNC(MONTHS\_BETWEEN(SYSDATE, TO\_DATE(b.due\_date, 'DD-MON-YY')))) AS months\_late,

   (COALESCE(SUM(n.price), 0)) \* 0.10 AS late\_fee,

   GREATEST(1, TRUNC(MONTHS\_BETWEEN(SYSDATE, TO\_DATE(b.due\_date, 'DD-MON-YY')))) \*

   ((COALESCE(SUM(n.price), 0)) \* 0.10) AS total\_late\_fee

   FROM bills b

   LEFT JOIN sales s ON b.customer\_id = s.customer\_id

   LEFT JOIN nonc\_inv n ON s.sale\_id = n.sale\_id

   WHERE b.status = 'Awaiting Payment'

   AND SYSDATE > b.due\_date

   GROUP BY

   b.billing\_id,

   b.late\_fee,

   b.due\_date; [↑](#footnote-ref-0)