Drew's Mattresses

Team 5

Thomas McNeill

Jason Rogers

Haley R. R. Mills

Executive Summary

Our development team was tasked with analyzing the needs, requirements, and potential deliverables for the client, Drew's Mattresses. The client, dealing with scaling issues, was considering the use of a relational database for meeting the needs of its day to day operations. A thorough understanding of the clients' expectations meant that clearly defined business rules and constraints would allow for a holistic approach for establishing both functional and nonfunctional requirements. A physical and logical model was introduced to establish a framework for implementing features that would help fulfill requirements the client had established as paramount.

The entities and relationships for these models were laid out in detail. A database system was then built and populated with input from previous and current customer acquisitions. Further input utilized for the database came from client company records. The SQL features outlined in this report reflect the necessity to both meet and exceed client demands, as well as, promote the functional use of the database for employees and customers. Additionally, certain triggers have been added to immediately update tables in the database to create ease of use for all parties involved.

The implementation of this database system for Drew's Mattresses will induce an environment that is more conditional for the CEO to focus on new avenues for expanding his business, rather than having to allocate resources for record keeping that has become both stale and stagnant. This advanced system will aid the client to remain competitive in an ever evolving digital age, and further meet the needs of his employees and consumers.

Introduction

Drew, the owner of Drew's Mattresses, started his small business almost five years ago - selling refurbished mattresses, box springs, bed frames and, to a lesser extent, other pieces of furniture. Over the years, his business has grown to include several employees, delivery trucks, and many storage units containing large amounts of merchandise procured from various suppliers he has formed business relationships with. While ad hoc accounting systems and spreadsheets have been good enough to track sales, procurements, inventory, and employee compensation until now, he has realized these disparate systems will present issues as his business continues to expand. With growth in mind, Drew would like to establish a secure, rules-based, consistent system of record for all aspects of his business in order to ensure reliable, available data conducive to an exponentially growing business.

Our team has created a relational database for Drew's Mattresses that will meet these needs. The database tracks transactions - customer sales and inventory procurement - inventory availability and storage unit location, customer and supplier information, employee information, hours and compensation, and business locations. The database records all aspects of Drew's business, and with it accounting records and inventory needs will be met with ease. The users of this database will be Drew - the CEO - for all aspects of business management and operations, employees for making sales, deliveries, marketing campaigns and restocking purposes, and customers, to inquire about available items, prices, and delivery fees.

BUSINESS RULES

Below are a list of business rules that must be enforced in the database in order to preserve data quality. These rules do not cover table relationships as specified in the section: *Description of Tables and Relationships*. These rules also exclude primary and foreign key constraints implied by table relationships.

Order Date cannot be null.
Order Status must be Pending, Canceled or Completed.
Order Line Number of Units must be greater than zero.
Delivery Fee must be determined by the formula: $10 flat fee + $1 per mile.
Delivery Miles must be greater than zero.
Delivery Driver License Number cannot be null.

Salesman Percent commission must be between 5-25%.
Vehicle Year must be prior or equal to the current system year plus one.
Vehicle odometer must be greater than zero.
Employee Hourly wage must be greater than or equal to minimum wage.
Employee Birth Date must be at least fourteen years prior to the system date.
Employee Hire Date must be prior or equal to the system date.
Inventory Item Condition must be New, Good or Worn.
Inventory Item Type must be one of the following: 'Bed Frame, California King', 'Bed Frame, Full', 'Bed Frame, King', 'Bed Frame, Queen', 'Bed Frame, Twin', 'Bed Frame, Twin XL', 'Box Spring, California King', 'Box Spring, Full', 'Box Spring, King', 'Box Spring, Queen', 'Box Spring, Twin', 'Box Spring, Twin XL', 'Chair, Arm', 'Chair, Desk', 'Chair, Dining', 'Chair, Other', 'Chair, Recliner', 'Chair, Wing', 'Desk', 'Dresser', 'Mattress, California King', 'Mattress, Full', 'Mattress, King', 'Mattress, Queen', 'Mattress, Twin', 'Mattress, Twin XL', 'Night Stand', 'Other', 'Sectional', 'Shelf', 'Sofa', 'Table, Coffee', 'Table, Dining Room', 'Table, Kitchen', 'Table, Other', or 'Wardrobe'.
Inventory Purchase Date must be prior or equal to the system date.
Inventory Purchase Amount must be greater than or equal to one.
Storage Unit Monthly Rent must be greater than zero.

USER REQUIREMENTS

Functional requirements:

1. Inventory Requirements
    - 1.1 The database stores data for all kinds of furniture, including mattresses, box spring, bedframes, shelves, night stands, couches, chairs, tables, coffee tables, and more. The item type attribute in the inventory type table will allow the employee to select from a drop-down of different furniture types.
    - 1.2 The inventory item table will store each piece of furniture that is purchased from vendors, stored in a unit and sold (historical data is maintained even if it has been sold).
2. Storage Unit Requirements
    - 2.1 Once an item is purchased from a vendor, the employee should be able to select from a drop down the storage unit number the furniture is stored in so it can be found by searching the database.
3. Customer Requirements

- 3.1 Customer information must be maintained so the customer can be contacted if they are expected to pick up furniture from the storefront if there is to be a delivery to the customer's address. If a delivery has been scheduled, the customer's address must be in the database.

4. Order/Purchase Requirements
    - 4.1 Because sales prices may be negotiated, the price of each item will only be recorded once the item has been sold to a customer.
    - 4.2 Item purchase price (from the supplier) will be entered with the piece of furniture as it is entered into the database when it is first supplied.
5. Employee Requirements
    - 5.1 Employees that act as salespeople must be identified so the amount of commission they earn can be queried/calculated from the database.
    - 5.2 Employees can either be salesman or delivery drivers.
6. Location Requirements
    - 6.1 Locations from which new inventory has been shipped must be identified so they can be recorded.
    - 6.2 Locations to which deliveries will be shipped must be identified so they can be recorded.
7. Supplier Requirements
    - 7.1 Suppliers must provide full names of the furniture sold.
    - 7.2 Suppliers must provide contact information and addresses so delivery drivers can pick up purchased items from the supplier's place of business if needed.
8. Delivery Requirements
    - 8.1 Delivery miles must be entered prior to payment so the delivery fee can be calculated.

9. Reporting Requirements
    - 8.1As mentioned previously, employees must record purchase prices, serial numbers, storage unit location, item type, sales prices, address, as well as the name of the employee who made the sale or delivery.

<u>Non functional requirements:</u>

1. Performance Requirements:

- 1.1. The system is easily accessible in order to ensure reporting can be done in any situation.
- 1.2. The system itself is guaranteed 24/7 up-time.

2. Operational Requirements:

- 2.1. The system will be accessible from across all locations and departments.
- 2.2. The system will be backed up regularly in both online and offline storage in order to make sure all data is preserved no matter what.

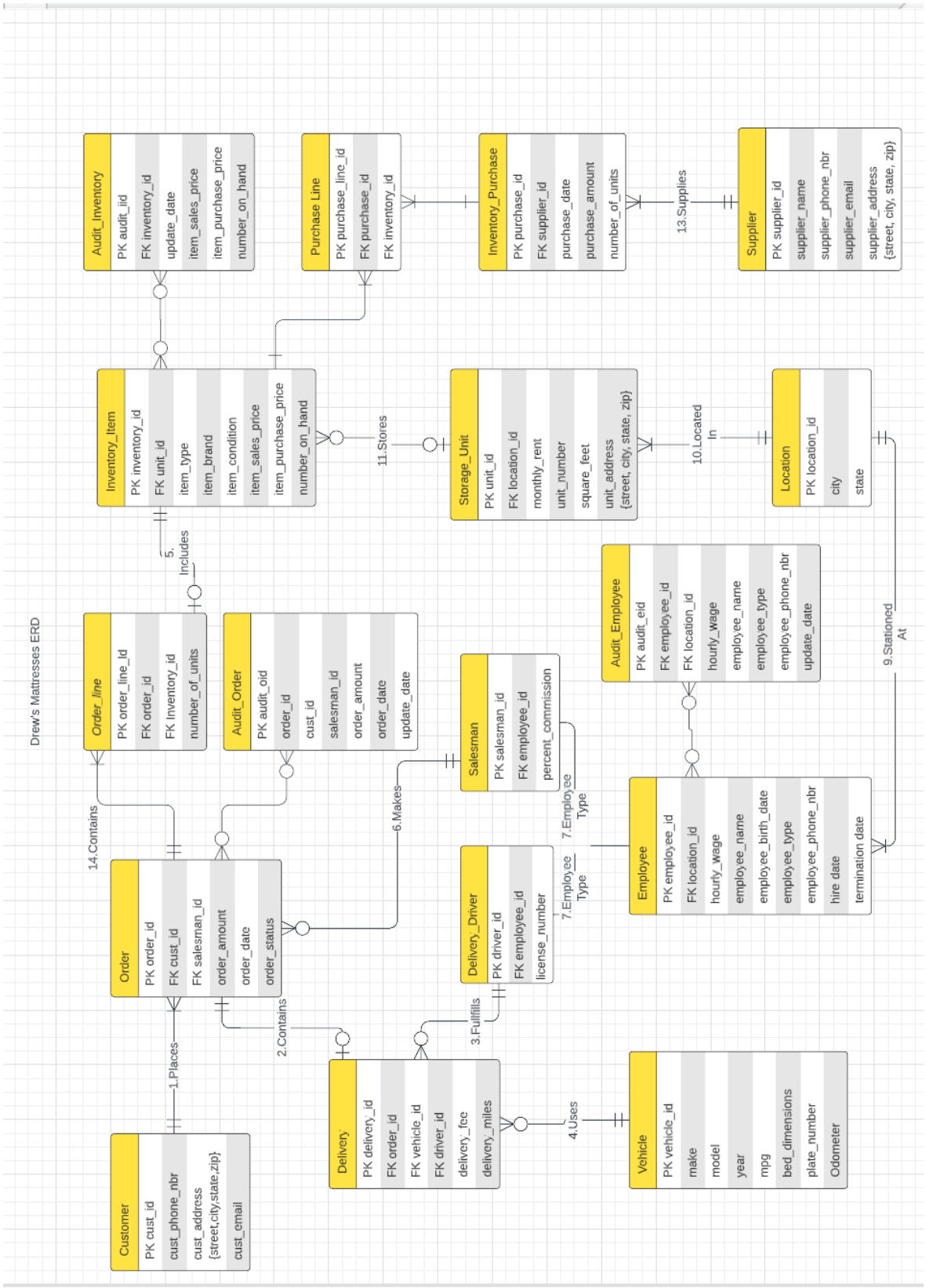
3. Security Requirements:

- 3.1. Physical security is ensured through a monitoring system that is overseen by the CEO.
- 3.2. The data reported will be saved somewhere inaccessible to unauthorized personnel.
- 3.3. The retrieval and reporting of data will always be traceable to the employee who requested or inputted it.

4. Cultural and Political Requirements:

- 4.1. USD is being used.
- 4.2. Administrators have access to capabilities that standard employees' accounts do not.
- 4.3. The syntax of the system will not be discriminatory in any way.
- 4.4. Local laws and regulations are not being violated.

Entity Relationship Diagram



Description of Tables and Relationships

Table Creation Code and Value Insertion Script: See attached file *Drews Mattresses Tables and Records 20221119.sql*

- Customer
    - Each line in this table indicates the information of a single *customer*. *Customer* phone numbers, addresses and email addresses are located in this table. *Customers* are referenced in the *Order* table. Each *customer* can place many *orders*. The primary key for this table is cust_id.
- Order
    - Each line in this table represents a single *order*, which is created by a *customer* and is facilitated by a *salesman*. Each order contains one to many *order lines*. A *delivery* is also associated with each *order*. Changes to this table are captured in the corresponding audit table. The primary key for this table is order_id and includes foreign keys cust_id and salesman_id for tables *Customer* and *Salesman*.
- Delivery
    - The delivery table contains the *delivery driver* and the *vehicle* for an *order*. The delivery fee and miles are captured for each *delivery*. Not all orders will have *deliveries*. The primary key for this table is delivery_id and includes foreign keys order_id, vehicle_id, and driver_id for tables *Order*, *Vehicle*, and *Delivery_Driver*.
- Vehicle
    - *Vehicle* make, model, year, miles per gallon, plate number, odometer number, and the dimensions of the *vehicle* bed are captured in this table for each *vehicle* used in a *delivery*. A *vehicle* can make many *deliveries* but a *delivery* can only have one *vehicle*. The primary key for this table is vehicle_id.
- Delivery Driver
    - A *delivery driver* is a type of *employee* that fulfills *deliveries*. The license number is captured in this table. Each *delivery* must have only one *delivery driver*, but a *delivery driver* can partake in many *deliveries*. The primary key for this table is driver_id and includes the foreign key employee_id for the *Employee* table.
- Employee

    - Each line in this table represents an individual *employee* for the mattress company. Each *employee* is stationed at a location and could potentially be a *delivery driver* or a *salesman*. Their personal information, as well as their hourly wage, hire date and termination date are captured in this table. Changes to this table are captured in the corresponding audit table. The primary key for this table is employee_id and includes foreign key location_id for the *Location* table.
- Salesman
    - A *salesman* is a type of *employee* who makes *orders*. They also receive a percent commission. Each *order* must have only one *salesman*, but a *salesman* could make many *orders*. The primary key for this table is salesman_id and includes foreign key employee_id for the *Employee* table.
- Inventory Item
    - A record in this table represents a single type of merchandise that the mattress store sells. The type, condition, sales price and purchase price are displayed here, as well as the number of units on hand. *Inventory* is kept in *storage units*, only one per inventory type. *Inventory* is included in the *order lines* when an order is created. Changes to this table are captured in the corresponding audit table. The primary key for this table is inventory_id and includes foreign key unit_id for the *Storage_Unit* table.
- Order Line
    - *Orders* for merchandise are broken out into *order lines* for each type of *inventory* the *customer* ordered, and this allows for multiple units of each type to be ordered, and the total to be calculated. Each *order line* contains one *inventory item*, and is associated with only one *order*. *Inventory items* can be used in many *orders*, and an order can have many *order lines*. The primary key for this table is order_line_id and includes foreign keys order_id and inventory_id for tables Order and *Inventory_Item*.
- Storage Unit
    - *Inventory* must be kept in *storage units*. The address, unit number, size, and monthly rent due is included in this table. The *storage unit* is located in a single *location*, but a *location* can have many *storage units*. The primary key for this table is unit_id and includes foreign key location_id for the *Location* table.
- Location

    - *Locations* are divided into city/state combinations to help identify the general area that *employees* or *storage units* were in. Each *location* can have many *employees*, but an *employee* can only be stationed in one *location*, this goes for *storage units* also. The primary key for this table is location_id.
- Supplier
    - The mattress company purchases *inventory items* from *suppliers*. This table contains details on the *supplier* such as name and contact information. *Suppliers* can fulfill many purchases, but an *inventory purchase* is associated with only one *supplier*. The primary key for this table is supplier_id.
- Inventory Purchase
    - To maintain the inventory levels, the mattress company must purchase new inventory from *suppliers*. These purchases are recorded in this table. The purchase date and amount are included, as well as the *supplier* for the purchase. The *inventory purchase* can contain many *purchase lines* which indicate which *inventory items* were being supplied, but each *purchase line* must only be associated with a single *inventory purchase*. The primary key for this table is purchase_id and includes foreign key supplier_id for the *Supplier* table.
- Purchase Line
    - Inventory purchases are broken out by inventory item, so that the purchase amount can be calculated. Each Purchase Line contains one inventory item, but inventory items can consist of many purchase lines. Each purchase line is associated with only one inventory purchase. The primary key for this table is purchase_line_id and includes foreign keys purchase_id and inventory_d for tables *Inventory_Purchase* and *Inventory_Item*.

Audit Tables - Explanation in the *Features* section

- Audit Orders
- Audit Employee
- Audit Inventory

Features

1. Trigger to add old employee record to employee audit table when a record is changed

Code:

```
-- Trigger to add old employee record to employee audit table when a record is changed

create or replace trigger employee_updates
after update on employee
for each row when (new.Location_ID <> old.Location_ID
                OR new.Hourly_Wage <> old.Hourly_Wage
                OR new.Employee_Name <> old.Employee_Name
                OR new.Employee_Type <> old.Employee_Type
                OR new.Employee_Phone_Nbr <> old.Employee_Phone_Nbr)
begin
insert into audit_employee values
   (aud_emp_seq.nextval,
   :old.Employee_ID,
   :old.Location_ID,
   :old.Hourly_Wage,
   :old.Employee_Name,
   :old.Employee_Type,
   :old.Employee_Phone_Nbr,
   :old.Hire_Date,
   :old.Termination_Date,
   sysdate);
end;

--Test
update employee set Location_ID = 3 where Employee_ID = 1;
select * from audit_employee;
```

Execution:

AUDIT_EID	EMPLOYEE_ID	LOCATION_ID	HOURLY_WAGE	EMPLOYEE_NAME	EMPLOYEE_TYPE	EMPLOYEE_PHONE_NBR	HIRE_DATE	TERMINATION_DATE	UPDATE_DATE
21	1	1	22	Eric Yorkie	Salesman	5773339847	09/12/2019	-	19-NOV-22 08.08.11.000000 PM

1 rows returned in 0.00 seconds Download

2. Procedure that allows an employee to update vehicle odometer for individual vehicles

Code:

```
-- Procedure that allows an employee to update vehicle odometer for individual vehicles

Create or replace procedure increase_odometer (v_id in number, odom_incr in
number) IS
v_odom number;
new_odom number;--declare variables
Begin
   select odometer into v_odom from vehicle where v_id = vehicle_id; --populate odometer from Vehicle Table based
on inputted value
   if v_odom > 0 then
      dbms_output.put_line('current odometer: ' || v_odom); --print current odometer
      update vehicle set odometer = v_odom + odom_incr where vehicle_id = v_id; --increase wage by inputted value
      new_odom := v_odom + odom_incr; --update value to print
      dbms_output.put_line('Odometer increased by: ' || odom_incr);
      dbms_output.put_line('New odometer: ' || new_odom);
      else
      dbms_output.put_line('no such vehicle found');
   end if;
End;

begin
increase_odometer(2, 2000); --call procedure
end;
```

Execution:

```
Bottom Splitter   Explain   Describe   Saved SQL   History

current odometer: 102000
Odometer increased by: 2000
New odometer: 104000

Statement processed.

0.00 seconds
```

```
select vehicle_id, odometer from vehicle where vehicle_id=2;
```

VEHICLE_ID	ODOMETER
2	104000

1 rows returned in 0.02 seconds Download

3. Procedure that allows a customer to determine the delivery fee (flat fee of $10 + $1 / mile)

Code:

```
-- Procedure that allows a customer to determine the delivery fee (flat fee of $10 + $1 / mile)

create or replace procedure get_delivery_fee(miles number) as
fee number;
begin
fee := miles + 10;
dbms_output.put_line('Delivery fee for this order: $'|| fee); end;

Begin get_delivery_fee(20); End;
```

Execution:

```
Delivery fee for this order: $30

Statement processed.


0.01 seconds
```

4. Procedure that allows employees to look up customers by city or zip code and returns contact information for targeted marketing campaigns

Code:

```
-- Procedure that allows employees to look up customers by city or zip code and returns contact information for
targeted marketing campaigns

create or replace procedure get_customer_info (areatype in varchar, searchstring in varchar) is
cursor cityc is select cust_email from customer where searchstring = cust_city; --city cursor that is only called if the
area type = 'city'
cursor zipc is select cust_email from customer where searchstring = cust_zip; --type cursor that is only called if the
area type = 'zip code'
custemail varchar(30); --declare variables
begin
if areatype = 'zip code' -- if statement runs different cursors based on the inputted type
then
open zipc; --opens cursor for type search
loop
fetch zipc into custemail; -- grabs the customer email from the cursor, based on the search string input
exit when zipc%notfound;
dbms_output.put_line('Customer Emails: ' || custemail); --prints customer emails
end loop;
elsif areatype = 'city' -- runs in type = brand
then
open cityc; --opens cursor for city search
loop
fetch cityc into custemail; -- grabs the emails from the cursor, based on the search string input
exit when cityc%notfound;
dbms_output.put_line('Customer Email: ' || custemail);--prints customer emails
end loop;
else
```

```
dbms_output.put_line('please enter "zip code" or "city" and then the search term'); -- if an invalid type is entered, error message is provided
end if;
end;
```

Execution:

```
begin get_customer_info('zip code', '23221'); end;
```

```
Customer Emails: Leah_Clearwater@yahoo.com
Customer Emails: bree_tanner@yahoo.com
Customer Emails: RileyBiers@yahoo.com
Customer Emails: Tyler__Crowley@yahoo.com
Customer Emails: ReneeflopDwyer@yahoo.com

Statement processed.
```

```
begin get_customer_info('city', 'Richmond'); end;
```

```
Customer Email: Leah_Clearwater@yahoo.com
Customer Email: bree_tanner@yahoo.com
Customer Email: RileyBiers@yahoo.com

Statement processed.


0.33 seconds
```

5. Procedure that allows employees to look up suppliers based on city or zip and returns contact information

Code:

```
-- Procedure that allows employees to look up suppliers based on city or zip and returns contact information

create or replace procedure get_supplier_info (areatype in varchar, searchstring in varchar) is
cursor cityc is select supplier_name, supplier_phone_nbr, supplier_email, supplier_street, supplier_city, supplier_state, supplier_zip from supplier where searchstring = Supplier_City; --city cursor that is only called if the area type = 'city'
cursor zipc is select supplier_name, supplier_phone_nbr, supplier_email, supplier_street, supplier_city, supplier_state, supplier_zip from supplier where searchstring = Supplier_Zip  ; --type cursor that is only called if the area type = 'zip code'
```

```
SupplierName     VARCHAR2 (50);
SupplierPhoneNbr VARCHAR2 (10);
SupplierEmail    VARCHAR2 (50);
SupplierStreet   VARCHAR(30);
SupplierCity     VARCHAR(20);
SupplierState    VARCHAR(2);
SupplierZip      VARCHAR(10); --declare variables
begin
if areatype = 'zip code' -- if statement runs different cursors based on the inputted type
then
open zipc; --opens cursor for type search
loop
fetch zipc into SupplierName, SupplierPhoneNbr, SupplierEmail, SupplierStreet, SupplierCity, SupplierState,
SupplierZip; -- grabs the supplier info from the cursor, based on the search string input
exit when zipc%notfound;
dbms_output.put_line('Supplier Name: ' || SupplierName || ' | Phone: ' || SupplierPhoneNbr || ' | Email: ' ||
SupplierEmail ||
                ' | Address: ' || SupplierStreet || ' ' || SupplierCity  || ' ' || SupplierState  || ' ' || SupplierZip); --prints supplier
emails
end loop;
elsif areatype = 'city' -- runs in type = brand
then
open cityc; --opens cursor for city search
loop
fetch cityc into SupplierName, SupplierPhoneNbr, SupplierEmail, SupplierStreet, SupplierCity, SupplierState,
SupplierZip; -- grabs the emails from the cursor, based on the search string input
exit when cityc%notfound;
dbms_output.put_line('Supplier Name: ' || SupplierName || ' | Phone: ' || SupplierPhoneNbr || ' | Email: ' ||
SupplierEmail ||
                ' | Address: ' || SupplierStreet || ' ' || SupplierCity  || ' ' || SupplierState  || ' ' || SupplierZip); --prints supplier
emails
end loop;
else
dbms_output.put_line('please enter "zip code" or "city" and then the search term');
-- if an invalid type is entered, error message is provided
end if;
end;
```

Execution:

```
begin get_supplier_info('zip code', '23234'); end;
```

```
Supplier Name: Bestie Pillows | Phone: 2008348833 | Email: bestie.pillows@gmail.com | Address: 52 Ferrum St Henrico VA 23234
Supplier Name: Bedknobs and Broomsticks | Phone: 3000000000 | Email: bnabs@gmail.com | Address: 79 VMI Rd Ashland VA 23234
Supplier Name: Keeps LLC | Phone: 4589228833 | Email: keeps@gmail.com | Address: 17 MarWash Cr Chesterfield VA 23234
Supplier Name: Yikes Limited | Phone: 7678909007 | Email: yikes@gmail.com | Address: 66 VUU St Richmond VA 23234
Supplier Name: Fluff and Stuff | Phone: 5734881133 | Email: flffnstff@gmail.com | Address: 403 Bridgewater Ct Richmond VA 23234

Statement processed.
```

```
begin get_supplier_info('city', 'Richmond'); end;
```

```
Supplier Name: Yikes Limited | Phone: 7678909007 | Email: yikes@gmail.com | Address: 66 VUU St Richmond VA 23234
Supplier Name: Fluff and Stuff | Phone: 5734881133 | Email: flffnstff@gmail.com | Address: 403 Bridgewater Ct Richmond VA 23234

Statement processed.


0.33 seconds
```

6. Trigger that calculates total order amount and updates order amount in order table before insert (must insert record into order_line table first)

Code:

```
-- Trigger that calculates total order amount and updates order amount in order table before insert
-- (must insert record into order_line table first)

create or replace trigger calculate_order_amount before insert on orders for each row
declare
x int;
begin
select sum(item_sales_price) into x from inventory_item i, order_line o
where i.inventory_id=o.inventory_id and order_id=:new.order_id group by order_id;
:new.order_amount := x;
end;
```

```
insert into Order_Line values('13', '7', '7', 2);
insert into orders values('7','3','5','100',CURRENT_TIMESTAMP, 'Pending');
select * from orders where order_id=7;
```

Execution:

*Changes order_amount from inserted value to actual calculation:

ORDER_ID	CUST_ID	SALESMAN_ID	ORDER_AMOUNT	ORDER_DATE	ORDER_STATUS
7	3	5	2100	20-NOV-22 11.56.17.449000 AM	Pending

7. trigger that adds inventory updates to audit_inventory table when item_sales_price, item_purchase_price or number_on_hand changes

Code:

```
-- trigger that adds inventory updates to audit_inventory table
-- when item_sales_price, item_purchase_price or number_on_hand changes
create sequence aud_inv_seq start with 1;

create or replace trigger inventory_updates
after update on inventory_item
for each row when (new.item_sales_price <> old.item_sales_price
                   OR new.item_purchase_price <> old.item_purchase_price
                   OR new.number_on_hand <> old.number_on_hand)
begin
insert into audit_inventory values
   (aud_inv_seq.nextval,
```

```
   :old.Inventory_ID,
   sysdate,
   :old.item_sales_price,
   :old.item_purchase_price,
   :old.number_on_hand);
end;
```

```
update inventory_item set item_sales_price = 3100 where inventory_id = 1;
select * from audit_inventory;
```

Execution:

AUDIT_IID	INVENTORY_ID	UPDATE_DATE	ITEM_SALES_PRICE	ITEM_PURCHASE_PRICE	NUMBER_ON_HAND
1	1	19-NOV-22 08.10.06.000000 PM	3000	2000	5

1 rows returned in 0.00 seconds Download

8. Trigger that calculates delivery fee and updates delivery fee in delivery table before insert

Code:

```
-- Trigger that calculates delivery fee and updates delivery fee in delivery table before insert

create or replace trigger update_delivery_fee
before insert or update on delivery
for each row
declare
x int;
begin
x := (:new.delivery_miles + 10);
:NEW.delivery_fee := x;
end;
```

```
delete from delivery;
```

```
insert all
into delivery values('1','5','2','9','10','32')
into delivery values('2','4','2','9','12','7')
into delivery values('3','3','1','2','30','40')
into delivery values('4','2','3','8','15','20')
into delivery values('5','1','5','2','15','82')
SELECT * FROM dual;
```

```
select delivery_id, delivery_fee, delivery_miles from delivery;
```

Execution:

*Inserted values for delivery_fee have been replaced with calculated values:

DELIVERY_ID	DELIVERY_FEE	DELIVERY_MILES
1	42	32
2	17	7
3	50	40

9. Procedure that allows employees to update the hourly wage of employees by an inputted amount

Code:

```
--Update hourly rate
Create or replace procedure increase_hourly_wage(emp_id in number,
wage_incr in
number) IS
E_wage number;
new_wage number;--declare variables
Begin
    select hourly_wage into e_wage from employee where emp_id =
employee_id; --populate wage from Employee Table based on inputted value
    if e_wage > 0 then
        dbms_output.put_line('current wage:' || e_wage);--print current
wage
        update employee set hourly_wage = e_wage + wage_incr where
employee_id =
        emp_id;--increase wage by inputted value
        new_wage := e_wage + wage_incr;--update value to print
        dbms_output.put_line('Wage increased by:' || wage_incr); --prints
inputted increase
        dbms_output.put_line('New wage:' || new_wage); -- prints new value
        else
        dbms_output.put_line('no such employee found'); -- error message
    end if;
End;

begin
increase_hourly_wage(3, 1);--call procedure
end;
```

```
select employee_id, hourly_wage from employee where employee_id = 3;
--check
```

Execution:

```
Begin
    select hourly_wage into e_wage from employee where emp_id = employee_id;
    if e_wage > 0 then
        dbms_output.put_line('current wage:' || e_wage);
        update employee set hourly_wage = e_wage + wage_incr where employee_id =
        emp_id;
        new_wage := e_wage + wage_incr;
        dbms_output.put_line('Wage increased by:' || wage_incr);
        dbms_output.put_line('New wage:' || new_wage);
        else
        dbms_output.put_line('no such employee found');
    end if;
End;

begin
increase_hourly_wage(3, 1);
end;
```

Results | Explain | Describe | Saved SQL | History

```
current wage:63
Wage increased by:1
New wage:64
```

rasmussenhr | info610_fall22 | en | Copy

–Check

```
        emp_id;
        new_wage := e_wage + wage_incr;
        dbms_output.put_line('Wage increased by:' || wage_incr);
        dbms_output.put_line('New wage:' || new_wage);
        else
        dbms_output.put_line('no such employee found');
    end if;
End;

begin
increase_hourly_wage(3, 1);
end;

select employee_id, hourly_wage from employee where employee_id = 3
```

Results | Explain | Describe | Saved SQL | History

EMPLOYEE_ID	HOURLY_WAGE
3	64

rasmussenhr | info610_fall22 | en | Copyright © 1999, 2021, Oracle and/or its affiliates. | Application Express 21.1.0

10. Procedure that allows an employee to update the sales commission percentage by an inputted amount

Code:

```
--Creates procedure that update sales commission based on given emp_id and
commission %
Create or replace procedure increase_commission(emp_id in number, com_incr
in number) IS
S_com number;
new_com number;--declare variables
Begin
    select percent_commission into s_com from salesman where emp_id =
salesman_id; --populate commission from Salesman Table based on inputted
value
    if s_com > 0 then
        dbms_output.put_line('Current commission:' || s_com);--print
current commission
        update salesman set percent_commission = s_com + com_incr where
salesman_id =
        emp_id;--increase wage by inputted value
        new_com := s_com + com_incr;--update value to print
        dbms_output.put_line('commission increased by:' || com_incr);--
prints inputted increase
        dbms_output.put_line('New commission:' || new_com); --prints new
commission
        else
        dbms_output.put_line('no such employee found'); -- error message
    end if;
End;

begin
increase_commission(6, 5);--call procedure
end;

select *
from salesman
where SALESMAN_ID = 6 --check
```

Execution:

```
Begin
    select percent_commission into s_com from salesman where emp_id = salesman_id; --populate commission from Salesman Table based on inputted value
    if s_com > 0 then
        dbms_output.put_line('Current commission:' || s_com);--print current commission
        update salesman set percent_commission = s_com + com_incr where salesman_id =
        emp_id;--increase wage by inputted value
        new_com := s_com + com_incr;--update value to print
        dbms_output.put_line('commission increased by:' || com_incr);
        dbms_output.put_line('New commission:' || new_com);
        else
        dbms_output.put_line('no such employee found');
    end if;
End;

begin
increase_commission(6, 5);--call procedure
end;
```

Results | Explain | Describe | Saved SQL | History

```
Current commission:30
commission increased by:5
New commission:35
```

rasmussenhr info610_fall22 en Copyright © 1999, 2021, Oracle and/or its affiliates.

– check

```
        dbms_output.put_line('New commission:' || new_com);
        else
        dbms_output.put_line('no such employee found');
    end if;
End;

begin
increase_commission(6, 5);--call procedure
end;

select *
from salesman
where SALESMAN_ID = 6 --check
```

Results | Explain | Describe | Saved SQL | History

SALESMAN_ID	PERCENT_COMMISSION
6	35

11. Procedure that allows a customer to input a brand or type of inventory and get back the inventory ID.

Code:

```
--creates a procedure that intakes the desired type of search (type or
brand), and returns all inventory IDs related to that search
create or replace procedure get_inv_id (invtype in varchar, searchstring
in varchar) is
cursor brandc is select inventory_id from inventory_item where
searchstring = item_brand; --brand cursor that is only called if the type
= 'brand', it uses the search string to pull brands
cursor typec is select inventory_id from inventory_item where searchstring
= item_type; --type cursor that is only called if the inventory type =
'type', it uses the search string to pull inventory types
invid number; --declare variables
begin
if invtype = 'type' -- if statement runs different cursors based on the
inputted type
then
open typec; --opens cursor for type search
loop
fetch typec into invid; -- grabs the inventory ID from the cursor, based
on the search string input
exit when typec%notfound;
dbms_output.put_line('Inventory IDs: ' || invid); --prints inventory ids
end loop;
elsif invtype = 'brand' -- runs in type = brand
then
open brandc; --opens cursor for brand search
loop
fetch brandc into invid; -- grabs the inventory ID from the cursor, based
on the search string input
exit when brandc%notfound;
dbms_output.put_line('Inventory IDs: ' || invid);--prints inventory ids
end loop;
else
dbms_output.put_line('please enter Type or Brand and then the search'); --
if an invalid type is entered, error message is provided
end if;
end;
```

Execution:

```
fetch brandc into invid;
exit when brandc%notfound;
dbms_output.put_line(invid);
end loop;
else
dbms_output.put_line('please enter Type or Brand and then the search');
end if;
end;

begin
get_inv_id('brand', 'K Brand');
end;
```

Results | Explain | Describe | Saved SQL | History

```
1
2

Statement processed.
```

0.00 seconds

rasmussenhr | info610_fall22 | en

12. Employee can enter inventory ID and get quantity on hand

Code:

```
-- Creates procedure that allows user to enter the inventory ID and get
the number of units on hand
create or replace procedure get_quantity (ivid in number) is
cursor c1 is select number_on_hand from inventory_item where inventory_id
= ivid;
quant number; -- declare variables
begin
open c1; -- open cursor
fetch c1 into quant;
dbms_output.put_line('Quantity on Hand is '||quant); -- print quantity on
hand
end;

Begin
get_quanty('1');-- Call procedure with given inventory ID
end;
```

Execution:

```
create or replace procedure get_quantity (ivid in number) is
cursor c1 is select number_on_hand from inventory_item where inventory_id = ivid;
quant number;
begin
open c1;
fetch c1 into quant;
dbms_output.put_line('Quantity on Hand is '||quant);
end;

begin
get_quantity(1);
end;
```

Results | Explain | Describe | Saved SQL | History

```
Quantity on Hand is 20

Statement processed.
```

0.00 seconds

rasmussenhr | info610_fall22 | en | Copyrig

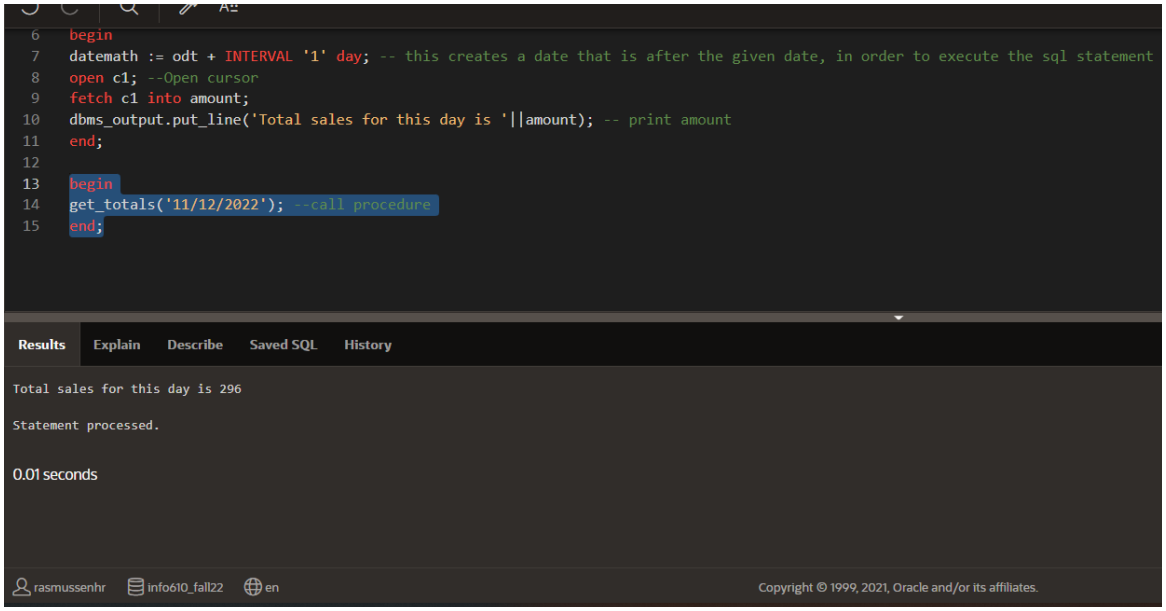
13. Procedure that allows an employee to enter a sales date and get back the total sales dollars for that day

Code:

```
-- Creates procedure that gets total order amount for a given day
create or replace procedure get_totals (odt in date) is
datemath date;
cursor c1 is select sum(order_amount) amt from orders where order_date <
datemath and order_date >= odt; -- declares cursor
amount number; -- declare variables
begin
datemath := odt + INTERVAL '1' day; -- this creates a date that is after
the given date, in order to execute the sql statement
open c1; --Open cursor
fetch c1 into amount;
dbms_output.put_line('Total sales for this day is '||amount); -- print
amount
end;
```

```
begin
get_totals('11/12/2022'); --call procedure
end;
```

Execution:



14. Procedure that allows an employee to get a list of all units/descriptions for an inputted storage unit

Code:

```
--creates procedure that gets info on a storage unit for an inputted unit
ID
create or replace procedure st_unit_desc(unitid in number)--input variable
is
locationid number;
monthlyrent number;
unitnumber number;
squarefeet number;-- declare local variables
begin
```

```
    select location_id, monthly_rent, unit_number, square_feet into
locationid, monthlyrent, unitnumber, squarefeet

    from storage_unit

    where unit_id = unitid; --SQL statements that gets the unit info

    if unitid > 0 then -- only runs if valid unit ID

        dbms_output.put_line('storage unit descriptions: Location: ' ||
locationid || ', Monthly Rent: '|| monthlyrent || ', Unit Number: ' ||
unitnumber || ', Square Ft: '|| squarefeet); --prints results

    end if;

end;


begin

st_unit_desc(2); -- runs procedure

end;
```

Execution:

```
end;

begin
st_unit_desc(2);
end;
```

Results | Explain | Describe | Saved SQL | History

```
storage unit descriptions: Location: 4, Monthly Rent: 300, Unit Number: 16, Square Ft: 300

Statement processed.
```

0.02 seconds

rasmussenhr | info610_fall22 | en | Copyri

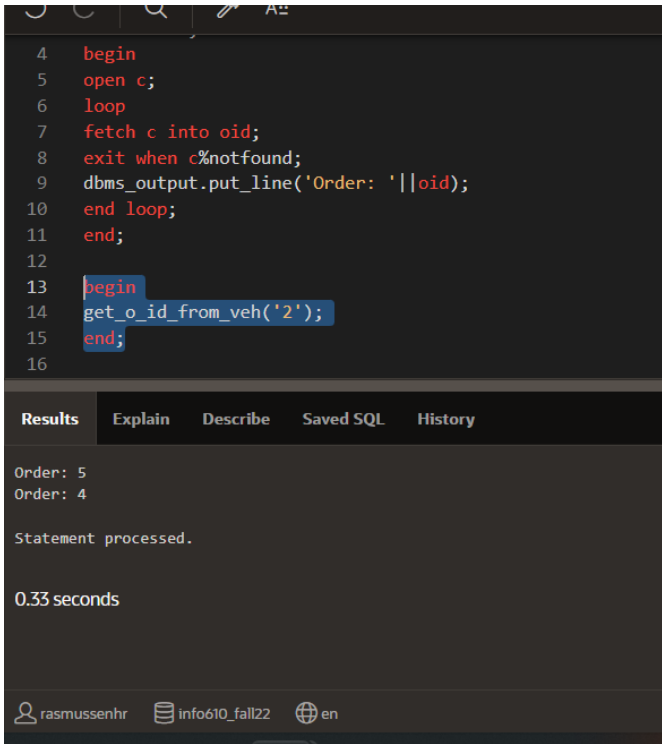
15. Procedure that allows an employee to enter a Vehicle ID and all get IDs for the orders that were delivered using that vehicle

Code:

```
--creates procedure that prints the order IDs for a given vehicle ID
create or replace procedure get_o_id_from_veh (vehid in number) is
cursor c is select order_id from vehicle i join delivery ii on
i.vehicle_id = ii.vehicle_id where i.vehicle_id = vehid;
oid number; --declare variable
begin
open c; --open cursor
loop
fetch c into oid;
exit when c%notfound; --exit loop when there are no more order IDs
dbms_output.put_line('Order: '||oid);--print order IDs
end loop;
end;

begin
get_o_id_from_veh('2'); --Run Procedure for given vehicle ID
```

```
end;
```

Execution:



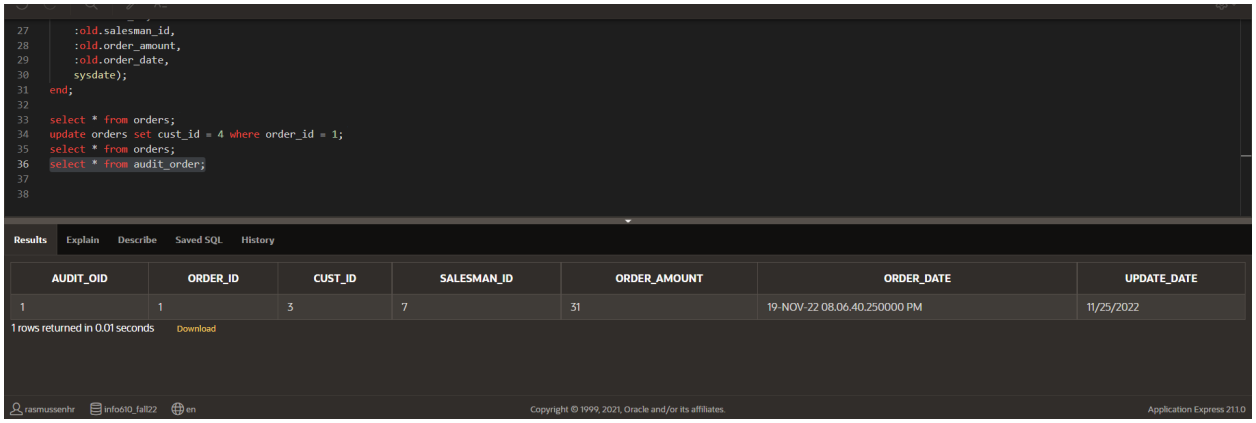
16. Trigger that adds order details if order is updated to audit table

Code:

```
--Trigger that adds order details if order is updated to audit table

 create sequence aud_ord_seq start with 1; --create sequence for audit
table


CREATE TABLE Audit_Order (--create audit table

Audit_OID CHAR (4) NOT NULL,

Order_ID CHAR (4) NOT NULL,

Cust_ID CHAR (4) NOT NULL,

Salesman_ID CHAR(4) NOT NULL,

Order_Amount DECIMAL (6, 2),
```

```
Order_Date TIMESTAMP,

Update_Date VARCHAR2 (10),

CONSTRAINT Aud_OID_PK PRIMARY KEY (Order_ID))

;


create or replace trigger order_updates --create trigger

after update on orders

for each row when (new.cust_id <> old.cust_id

        OR new.salesman_id <> old.salesman_id

        OR new.order_amount <> old.order_amount

        OR new.order_date <> old.order_date) --runs when order is updated

begin

insert into audit_order values

    (aud_ord_seq.nextval,

    :old.order_id,

    :old.cust_id,

    :old.salesman_id,

    :old.order_amount,

    :old.order_date,

    sysdate);--updates audit table with old values

end;


 --check

select * from orders;

update orders set cust_id = 4 where order_id = 1;

select * from orders;

select * from audit_order;
```

Execution:



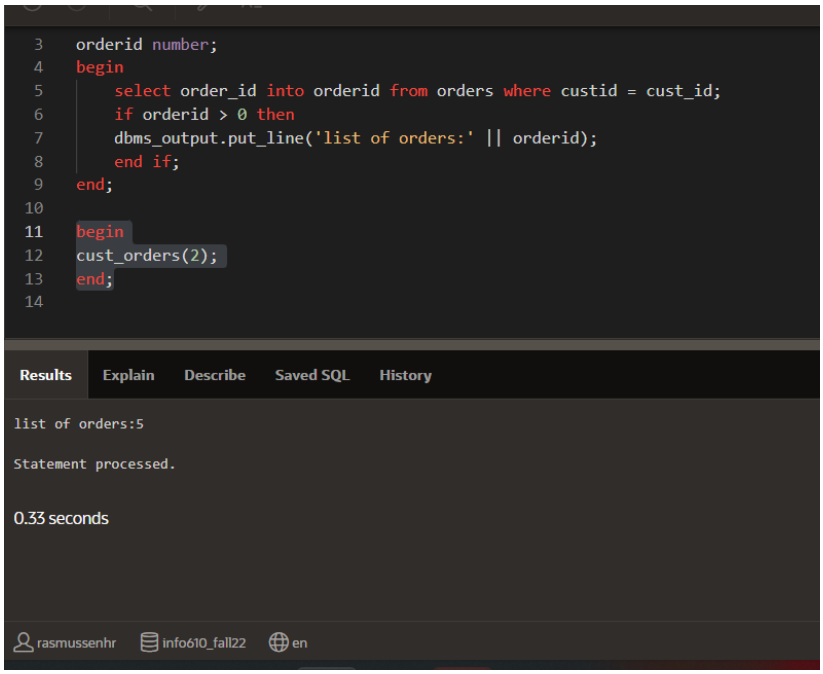
17. Procedure that allows an employee to enter the customer ID and get back the list of orders that customer had placed

Code:

```
--create procedure that gets orderids for inputted customer ID

create or replace procedure cust_orders(custid in number)--input variables

is

orderid number;--declare local variables

begin

    select order_id into orderid from orders where custid = cust_id; --SQL
statement that gets order ID

    if orderid > 0 then --only runs with valid order ID

    dbms_output.put_line('list of orders:' || orderid); --outputs list of
order_ids

    end if;

end;


 begin

cust_orders(2); --run procedure
```

```
end;
```

Execution:



18. Procedure that allows an employee to increase the sales price of an inventory item

Code:

```
--Creates procedure that update sales price based on given inventory ID
and increase %

Create or replace procedure increase_salesprice(inv_id in number,
price_incr in number) IS

S_price number;

new_price number;--declare variables

Begin

    select item_sales_price into s_price from inventory_item where inv_id
= inventory_id; --populate price from inventory Table based on inputted
value

    if s_price > 0 then
```

```
        dbms_output.put_line('Current Sales Price:' || s_price);--print
current price

        update inventory_item set item_sales_price = s_price +
(s_price*price_incr) where inv_id = inventory_id;--increase price by
inputted value

        new_price := s_price + (s_price*price_incr);--update value to
print

        dbms_output.put_line('Price increased by:' || price_incr);--
prints inputted increase

        dbms_output.put_line('New Price:' || new_price); --prints new
price

        else

        dbms_output.put_line('no such Inventory ID found'); -- error
message

    end if;

End;


begin

increase_salesprice(4, .05);--call procedure

end;
```

Execution:

```
begin
increase_salesprice(4, .05);--call procedure
end;
```

Results | Explain | Describe | Saved SQL | History

```
Current Sales Price:3675
Price increased by:.05
New Price:3858.75

Statement processed.
```

0.01 seconds

rasmussenhr | info610_fall22 | en

Peer Evaluation

We all equally contributed to the project.