Name: Cushing, Bradley
Date: 12/4/24
NYU ID: N10695516
Course Section Number: CSCI-GA-2433-001
Project Part #3 Report

Files included

- cushing_p3_fa24_physical_model.sql
- optimization/indexing.sql
- optimization/partitioning.sql
- prediction/model.pkl
- prediction/train-model.py
- prediction/predict-risk.py
- prediction/external-risk-actual.csv
- prediction/external-risk-random.csv

Physical Database Design

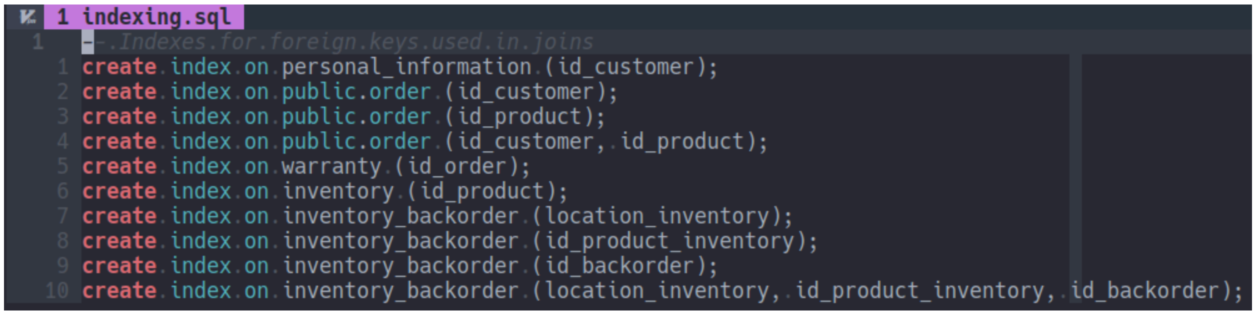
The below sections outline the physical database design including optimization like indexing and partitioning that were made. All indexes and partitions are specified in the single physical model file “cushing_p3_fa24_physical_model.sql” which includes all of the DDL code for the database. I have also included two separate files which have the indexing and partitioning code separated so it’s easier to see exactly what optimizations were done. The last two sections explain how the local and remote database environments were setup.

Perform and document

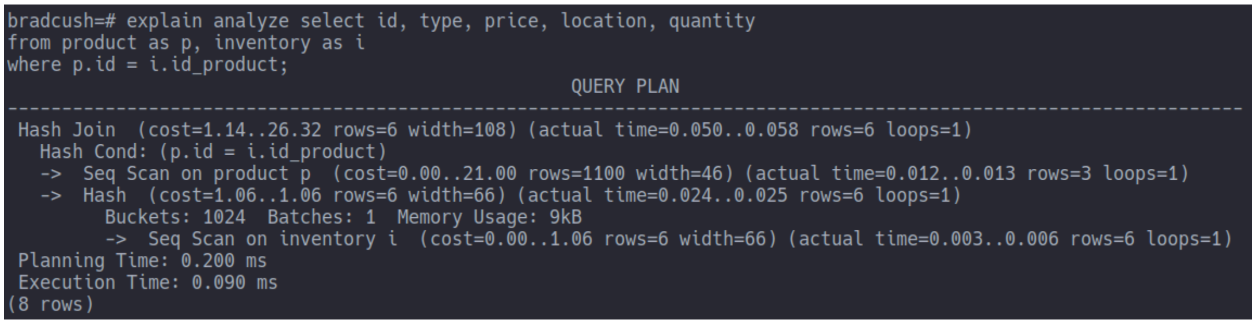
Indexing

It’s natural to create indexes for foreign_keys in tables that we expect to join. This is because in the use cases defined below we expect to join tables based on their primary keys and related foreign keys. Since PostgreSQL automatically creates indexes for primary keys, we only create indexes manually for foreign keys. I’ve done this only for the tables related sales since these will be the largest tables and will be queried the most often.

For reference, please see the file “optimization/indexing.sql” which contains all index creation statements. These tables are order, warranty, personal_information, inventory, and inventory_backorder. Note that the product table is not included since it contains no foreign keys. I’ve included two screenshots showing create index statements and the output of a query plan for a join on product and inventory using one of the below indexes.



```sql
-- Indexes for foreign keys used in joins
create index on personal_information (id_customer);
create index on public.order (id_customer);
create index on public.order (id_product);
create index on public.order (id_customer, id_product);
create index on warranty (id_order);
create index on inventory (id_product);
create index on inventory_backorder (location_inventory);
create index on inventory_backorder (id_product_inventory);
create index on inventory_backorder (id_backorder);
create index on inventory_backorder (location_inventory, id_product_inventory, id_backorder);
```

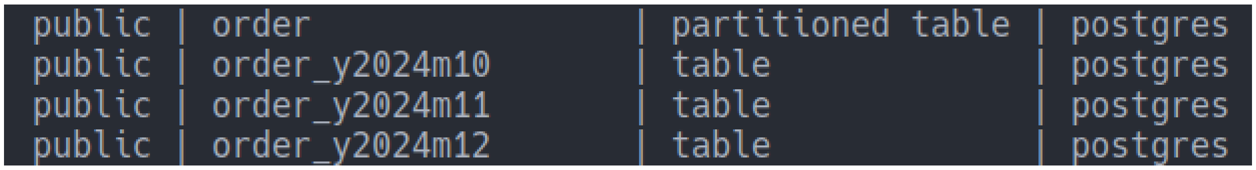


```
bradcush=# explain analyze select id, type, price, location, quantity
from product as p, inventory as i
where p.id = i.id_product;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1.14..26.32 rows=6 width=108) (actual time=0.050..0.058 rows=6 loops=1)
   Hash Cond: (p.id = i.id_product)
   ->  Seq Scan on product p  (cost=0.00..21.00 rows=1100 width=46) (actual time=0.012..0.013 rows=3 loops=1)
   ->  Hash  (cost=1.06..1.06 rows=6 width=66) (actual time=0.024..0.025 rows=6 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on inventory i  (cost=0.00..1.06 rows=6 width=66) (actual time=0.003..0.006 rows=6 loops=1)
 Planning Time: 0.200 ms
 Execution Time: 0.090 ms
(8 rows)
```

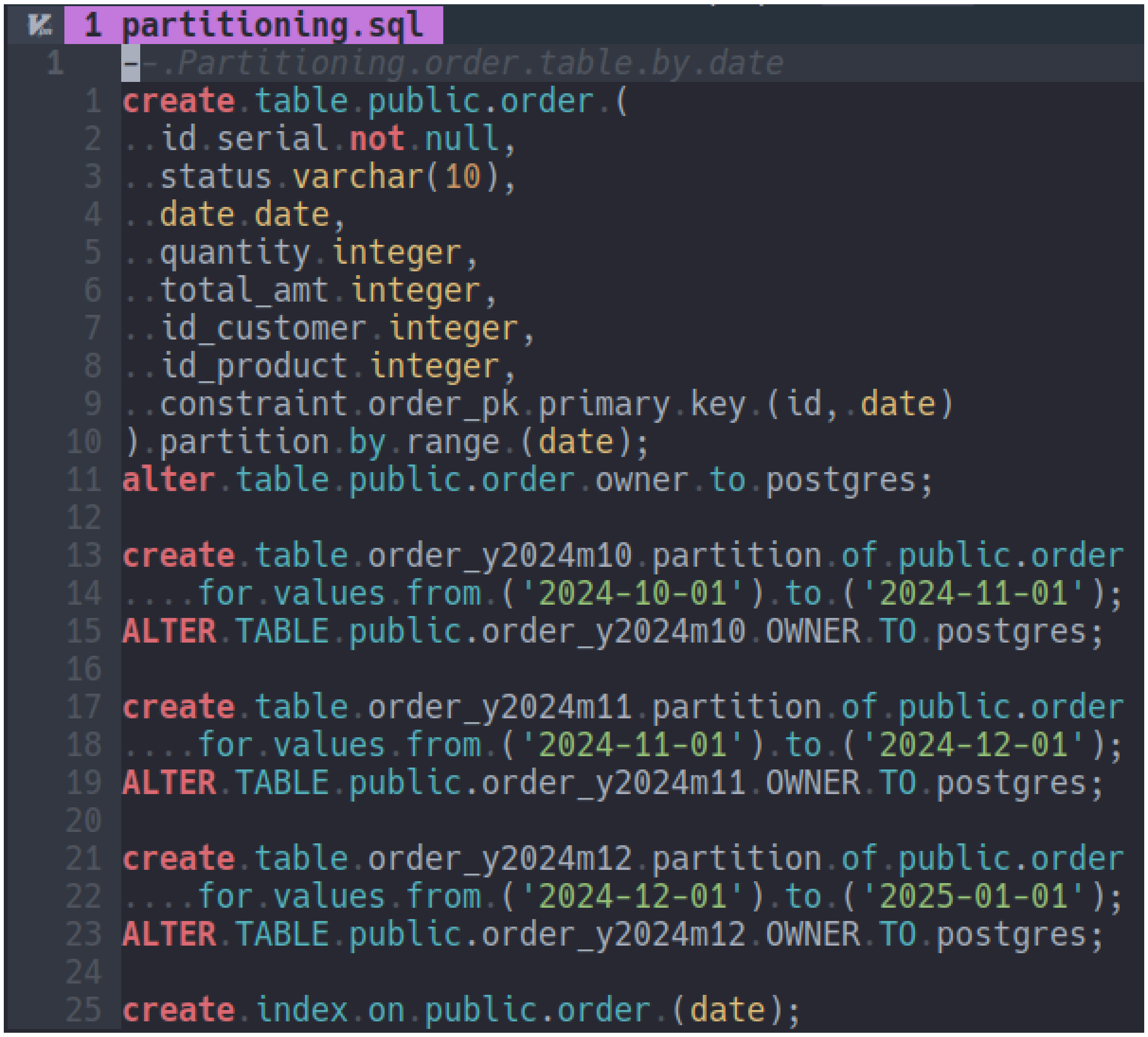
Partitioning

Given the assumption that the order table will be the most heavily used table and contain the most entries it makes the most sense to partition this table. Partitioning orders will allow us to support more concurrency for reads and writes on this data. It will also allow our queries to run much faster since we'll only need to look in some of the partitions for certain queries. We've used the date of the order as the column we want to use to partition which is a natural choice. Each partition will only include orders for a given month. We also create an index into each partition so that we can quickly access each table.

PostgreSQL supports declarative partitioning which was used. I've created partitions for the 3 months of data that exists. Any future partitions would need to be created automatically each month using a script for example. I've included a screenshot below of the logged tables from my local database instance and the code used to create the partition.



```
 public | order              | partitioned table | postgres
 public | order_y2024m10     | table             | postgres
 public | order_y2024m11     | table             | postgres
 public | order_y2024m12     | table             | postgres
```



1 partitioning.sql

```sql
-- Partitioning order table by date
create table public.order (
  id serial not null,
  status varchar(10),
  date date,
  quantity integer,
  total_amt integer,
  id_customer integer,
  id_product integer,
  constraint order_pk primary key (id, date)
) partition by range (date);
alter table public.order owner to postgres;

create table order_y2024m10 partition of public.order
    for values from ('2024-10-01') to ('2024-11-01');
ALTER TABLE public.order_y2024m10 OWNER TO postgres;

create table order_y2024m11 partition of public.order
    for values from ('2024-11-01') to ('2024-12-01');
ALTER TABLE public.order_y2024m11 OWNER TO postgres;

create table order_y2024m12 partition of public.order
    for values from ('2024-12-01') to ('2025-01-01');
ALTER TABLE public.order_y2024m12 OWNER TO postgres;

create index on public.order (date);
```

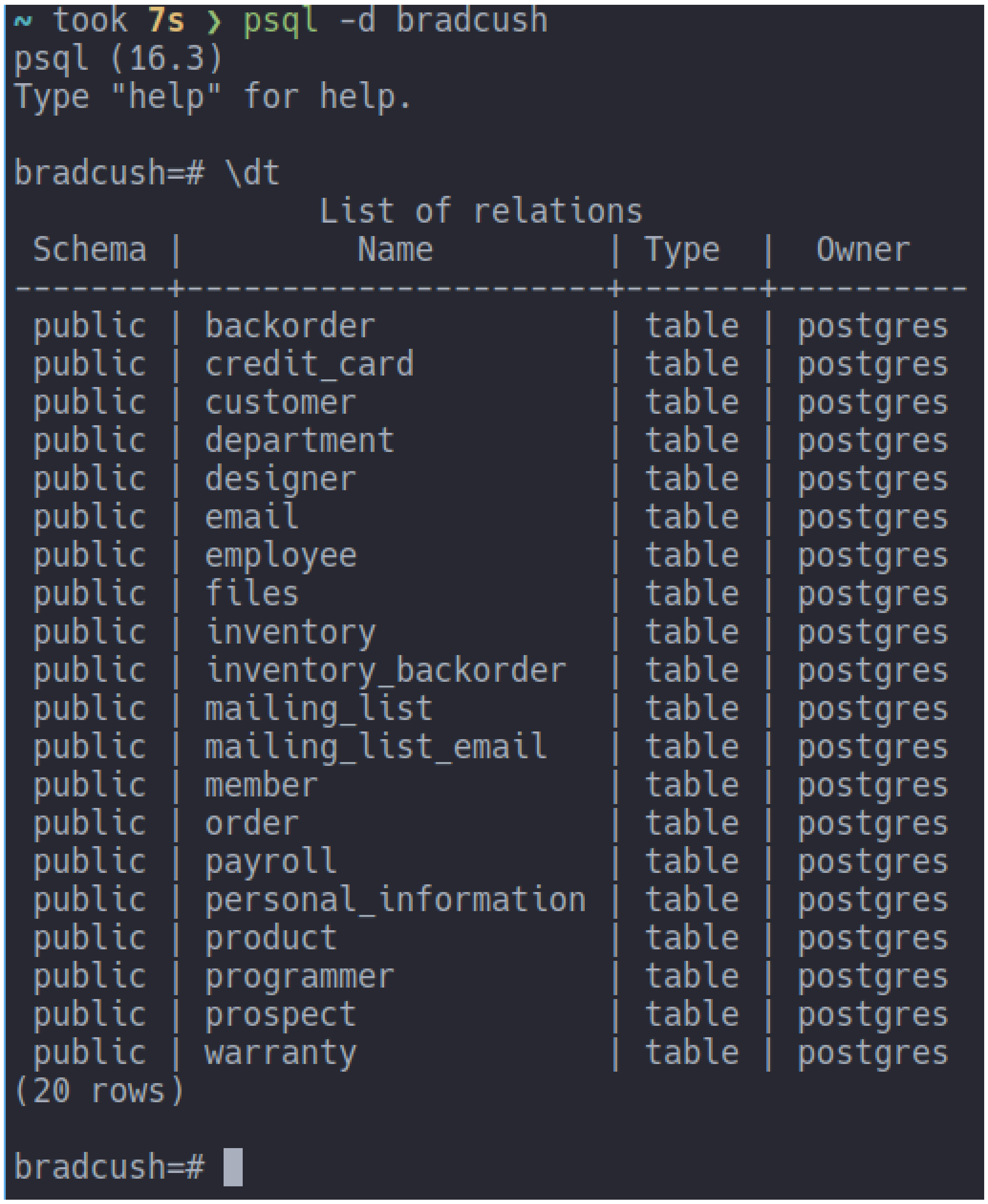
Deployment locally and remotely

As stated previously, I've chosen to use PostgreSQL for my RDBMS. I've gone ahead and deployed this database both locally for testing purposes and remotely using Heroku. Heroku offers a comprehensive solution for testing and deploying full-stack applications which can consist of a Python/Flask server and an RDBMS (PostgreSQL) database. As this is the technology I decided to go with on part 2 of the project, Heroku felt like a good fit. I've included screenshots of my database running both locally and remotely below.

Local database instance

I've deployed the Circuit Blocks, Inc database locally using PostgreSQL. In the screenshot below you can see the 20 tables which make up the database and map directly to the E-R model specified in part 1 and modified in part 2.

Heroku allows a locally running application to interact with some database of the same name as the primary user of the system, which in this case is marked as “bradcush”. My workflow entails running the application locally and testing changes before I deploy them to the remote server. I’m also able to replicate my local database on the remote instance when needed.

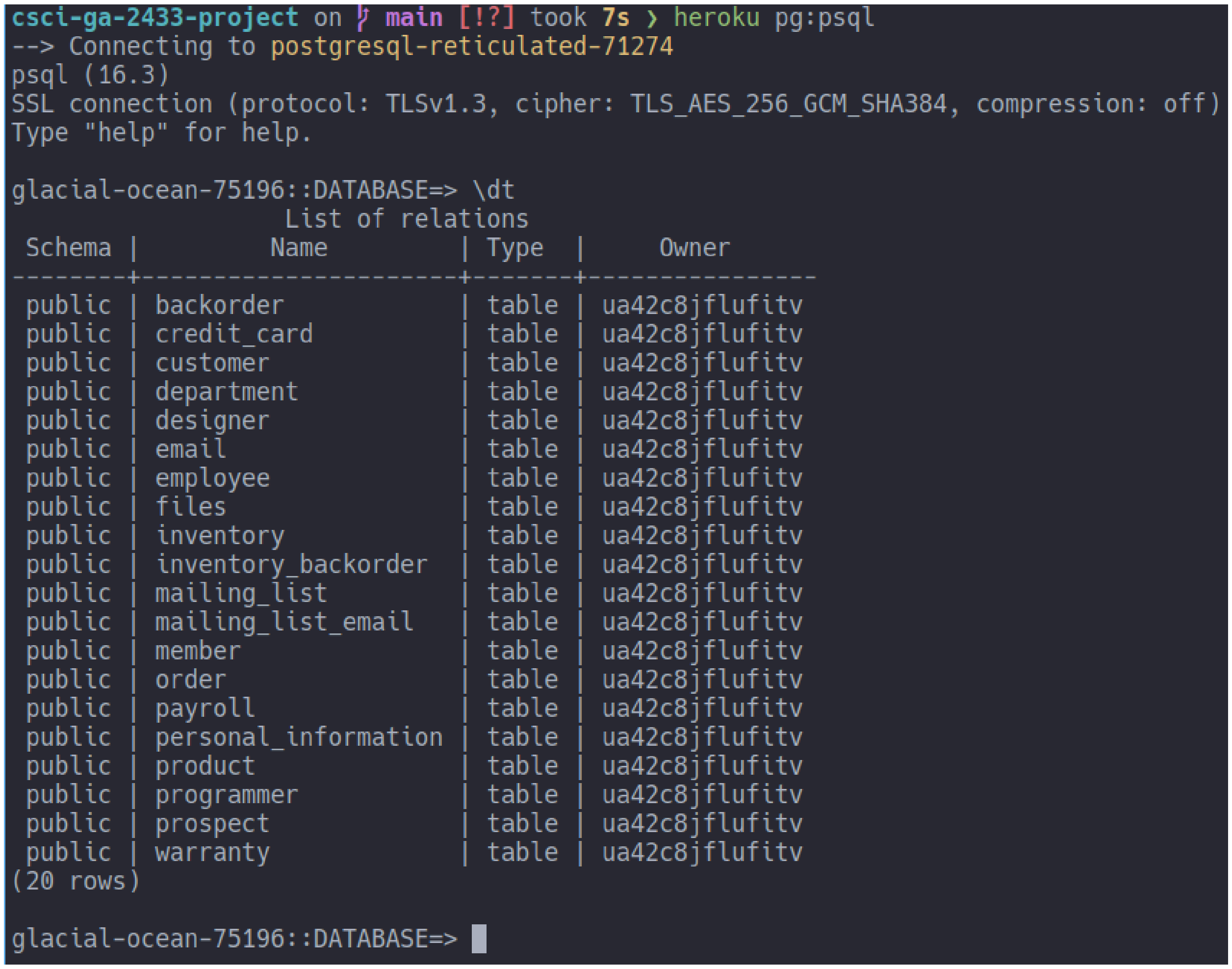


```
~ took 7s ❯ psql -d bradcush
psql (16.3)
Type "help" for help.

bradcush=# \dt
                 List of relations
 Schema |         Name          | Type  |  Owner
--------+-----------------------+-------+----------
 public | backorder             | table | postgres
 public | credit_card           | table | postgres
 public | customer              | table | postgres
 public | department            | table | postgres
 public | designer              | table | postgres
 public | email                 | table | postgres
 public | employee              | table | postgres
 public | files                 | table | postgres
 public | inventory             | table | postgres
 public | inventory_backorder   | table | postgres
 public | mailing_list          | table | postgres
 public | mailing_list_email    | table | postgres
 public | member                | table | postgres
 public | order                 | table | postgres
 public | payroll               | table | postgres
 public | personal_information  | table | postgres
 public | product               | table | postgres
 public | programmer            | table | postgres
 public | prospect              | table | postgres
 public | warranty              | table | postgres
(20 rows)

bradcush=#
```

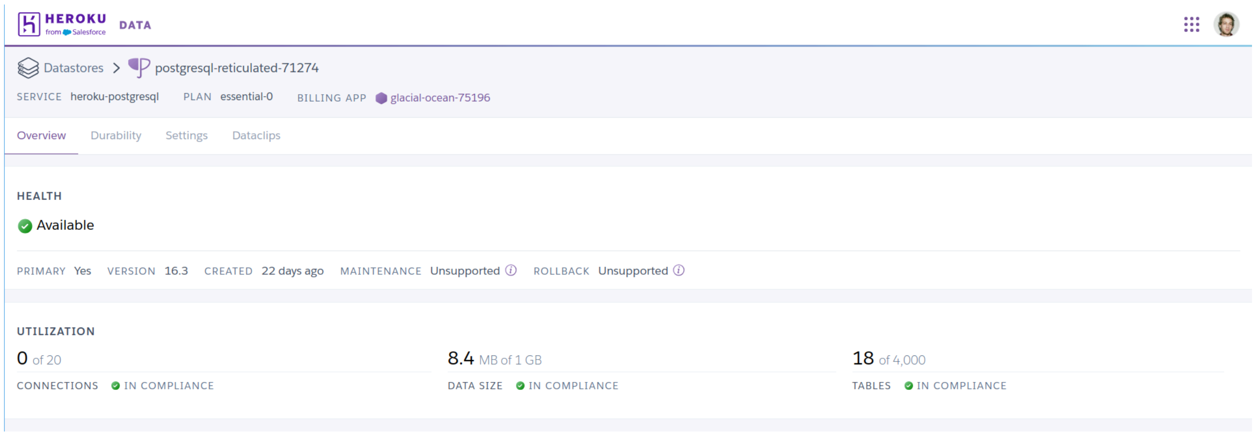
Remote database instance

The below screenshot shows a connection from my local machine to the remote database instance for Circuit Blocks, Inc. When the application is deployed remotely it automatically connects to this database instead of the local instance above when run locally.

You can see the remote instance has the same tables as the local database, following the E-R model specified in earlier parts of the project, but the owner is different as it's specific to the remote machine. As mentioned before, this database is managed remotely using Heroku. I've also included a screenshot of the overview panel which shows the overview information directly from Heroku for the same database instance.



```
csci-ga-2433-project on ᚠ main [!?] took 7s ❯ heroku pg:psql
--> Connecting to postgresql-reticulated-71274
psql (16.3)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

glacial-ocean-75196::DATABASE=> \dt
                   List of relations
 Schema |         Name         | Type  |     Owner
--------+----------------------+-------+----------------
 public | backorder            | table | ua42c8jflufitv
 public | credit_card          | table | ua42c8jflufitv
 public | customer             | table | ua42c8jflufitv
 public | department           | table | ua42c8jflufitv
 public | designer             | table | ua42c8jflufitv
 public | email                | table | ua42c8jflufitv
 public | employee             | table | ua42c8jflufitv
 public | files                | table | ua42c8jflufitv
 public | inventory            | table | ua42c8jflufitv
 public | inventory_backorder  | table | ua42c8jflufitv
 public | mailing_list         | table | ua42c8jflufitv
 public | mailing_list_email   | table | ua42c8jflufitv
 public | member               | table | ua42c8jflufitv
 public | order                | table | ua42c8jflufitv
 public | payroll              | table | ua42c8jflufitv
 public | personal_information | table | ua42c8jflufitv
 public | product              | table | ua42c8jflufitv
 public | programmer           | table | ua42c8jflufitv
 public | prospect             | table | ua42c8jflufitv
 public | warranty             | table | ua42c8jflufitv
(20 rows)

glacial-ocean-75196::DATABASE=>
```



SERVICE heroku-postgresql PLAN essential-0 BILLING APP glacial-ocean-75196

HEALTH

Available

PRIMARY Yes VERSION 16.3 CREATED 22 days ago MAINTENANCE Unsupported ⓘ ROLLBACK Unsupported ⓘ

UTILIZATION

0 of 20	8.4 MB of 1 GB	18 of 4,000
CONNECTIONS ● IN COMPLIANCE	DATA SIZE ● IN COMPLIANCE	TABLES ● IN COMPLIANCE

Connecting to use cases

We can define one overarching use case which can be thought of as being comprised of many sub-use cases. This overarching use case represents some large flow a user can take from beginning to end which involves interaction with various parts of the system.

We will specify the steps for this general use case and denote the sub-use cases in bold. In our application we plan to build all sub-use cases independently. This will allow a user to execute any subset of them, in any order, including the entire general use case described here.

Overarching use case

The final outcome of this general use case is the purchasing of a warranty for an order. Multiple steps are required for this to be possible. A product needs to be in inventory, a user needs to become a customer by giving some information including a credit card, a customer needs to place an order, and then finally the user can purchase a warranty for that order. We outline these steps below as the overarching use case involving a user and employee.

- Employee adds product information
- Employee adds inventory information
- User places an order for a product
  - User provides customer information
  - User provides credit card information
  - User provides order information
- User purchases warranty for order
  - User submits personal information for warranty discount
  - User provides warranty information
- User views order with warranty information

Note that for the above general use case, there is a step where the user is allowed to submit personal information for a warranty discount. This is where we use our trained model which is explained later in this document to make a prediction used to calculate the warranty price.

Sub-use cases explained

We've broken down each of the above steps further into what we will call sub-use cases below. We plan to support most of these use cases to allow for the full flow above where some steps are performed by an employee and others by the user/customer. For those use cases we don't implement we will pre-populate the necessary tables with data to enable the others.

Employee adds product information

- Employee given product type and price
- Employee adds product using form

Employee adds inventory information

- Employee given location, quantity, and id_product

- Employee adds inventory data using form

User places an order for product

- User enters customer information using form
- User enters credit card information using form
- User enters order information using form
    - This quantity, id_customer, and id_product
    - We can allow the email instead of id_customer
- User submits form and order is created

User purchases warranty for order

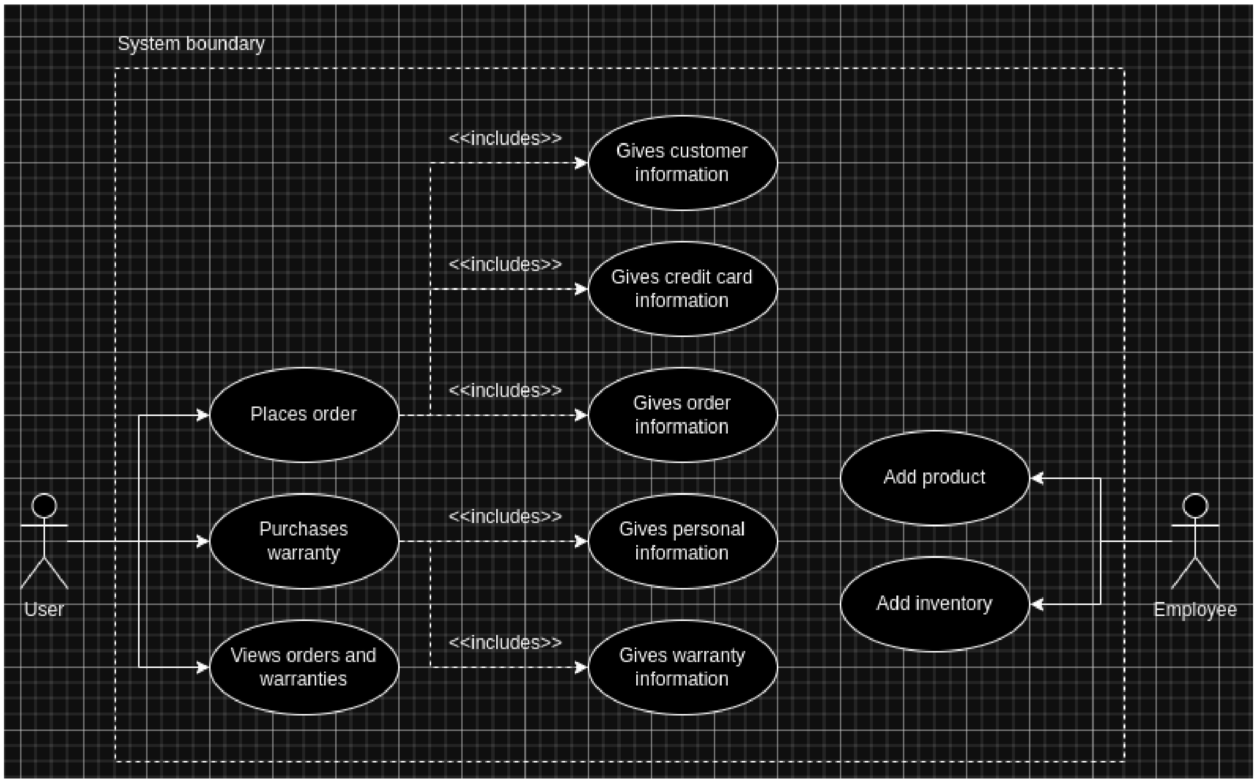
- User enters additional personal information
- User enters id_order for warranty purchase
    - Note price is a percentage of the order chosen
    - Determined by model prediction on personal information
        - No personal information: 20% of order price
        - Personal information (risky): 15% of order price
        - Personal information (not risk): 10% of order price
- User submits form and warranty is created

User views order with warranty information

- User enters id_customer or email address
- Views table with orders that have warranties

Use case diagram

The below use case diagram outlines the use cases above contained within the system boundary. All sub-use cases that are part of the overarching use case are covered.



ML Model Creation

Refine and complement use cases

To recap a bit from part 2 of this project, we are going to be using external data which contains anonymized personal information of people correlated with a risk score. The idea is that we can train a model to make a risk prediction for new users purchasing product warranties. Users will choose to give us some personal information in exchange for a discount on their warranty. We will assign a discount based on if our prediction of the user is risky or not.

Warranties are order specific and are priced at 20% of the original product order price. Anyone who submits additional, personal information will only have to pay 15% of the order price (we'll use this information to inform marketing campaigns which is why we value it) and those we predict as not risky will only have to pay 10%. When someone uses a warranty they get a full replacement for that particular order.

Price per warranty

- (submits no information) 20% of original order
- (submits information but risky) 15% of original order
- (submits information but not risky) 10% of original order

Identified use case

- User decides to purchase a warranty for their product
- User is given option to submit personal information for a discount
- User enters the id_order they would like to purchase a warranty for
- User inputs the 5 requested fields of personal information
- Application layer fetches the latest model from analytics database
- Application layer runs prediction model to determine associated risk
- Given the risk prediction we charge them for the warranty

We keep most of the details from the original use case for external data from part 2 which still fits well into our design and for what we want to get out of it. The use of this external data is important to the business for a few reasons we list below.

Personal information for marketing

We collect personal information from customers which we will be able to leverage for marketing campaigns later on and which will help us to know how we can target. We give users an incentive to submit this information since we discount their warranty no matter the prediction.

Reducing warranty risk

If we predict someone purchasing a warranty to be risky they will be charged more than someone who is predicted not to be a high risk. This allows us to charge a higher price per warranty for those we expect are more likely to request a replacement product versus those that are more likely not to. The idea is that this difference in price allows us to make sure we don't lose money when insuring risky customers. We can also tweak the percentage price we charge for each group as time goes on depending how our bottom line is affected.

Select and train ML model

We've trained a model using Logistic Regression to predict warranty risk. Logistic Regression is the best fit because our data includes a risk column that we want to be able to predict as accurately as possible given the other data points in the data set. The fields we use for prediction are Age, Kids_count, Pets_count, Siblings_count, and Income amount. We deem these as good predictors for whether or not the person is risky, which is someone we consider more likely to request a replacement product.

Note that the intuition that led to this data being used and a model being trained is that those with more kids and pets will have higher risk and that those with more siblings and higher income will have lower risk. I've referenced a list of files below in the prediction folder that are included as part of the submitted ZIP file. These files include the model itself as a PKL file, a script to train the model using the external risk data, and a script testing risk prediction which uses the output model from the previous step.

- prediction/model.pkl
- prediction/train-model.py

- prediction/predict-risk.py
- prediction/external-risk-actual.csv

Prediction results

The trained model is roughly 90% accurate when predicting against a subset of the original data used to train. This shows that the training yielded a model that can predict with high accuracy whether someone purchasing a warranty will be a risk to the business or not.

Big data platform and data lake

In the previous part 2, aside from anonymized personal information, I collected title and wage data from the Department of Labor. This data could be used in a similar fashion where yearly we train a model to predict the salary of an employee and use this prediction to help inform our change in salary calculation for employees within the company. Informing the decision on how to change the pay of our employees keeps us competitive in the market and our employees happy. Conversely, if the market is paying less we can consider adjusting our costs to be aligned with the market so we aren't overpaying and spending money unnecessarily.

Below I discuss how we can react to changes in the anonymized personal information we've collected which informs the prices we set for warranties. In the following section I go into more details about the reference architecture and how it scales with sources of data.

Reacting to data changes

The price we charge for a warranty is a percentage of the order price. Remember that it is either 20%, 15%, or 10% depending on whether personal information is submitted and if so what the predicted risk for that customer warranty is based on it.

We could imagine that if we see changes in the data set in which we train the model this might need to impact the percentages we set and how much we charge for warranties. Right now prices are hardcoded in the Python/Flask application. To make our design more flexible, we could opt to move them into a separate price table which we update based on the percentage of overall risk we see in the external data set.

Example warranty_price table

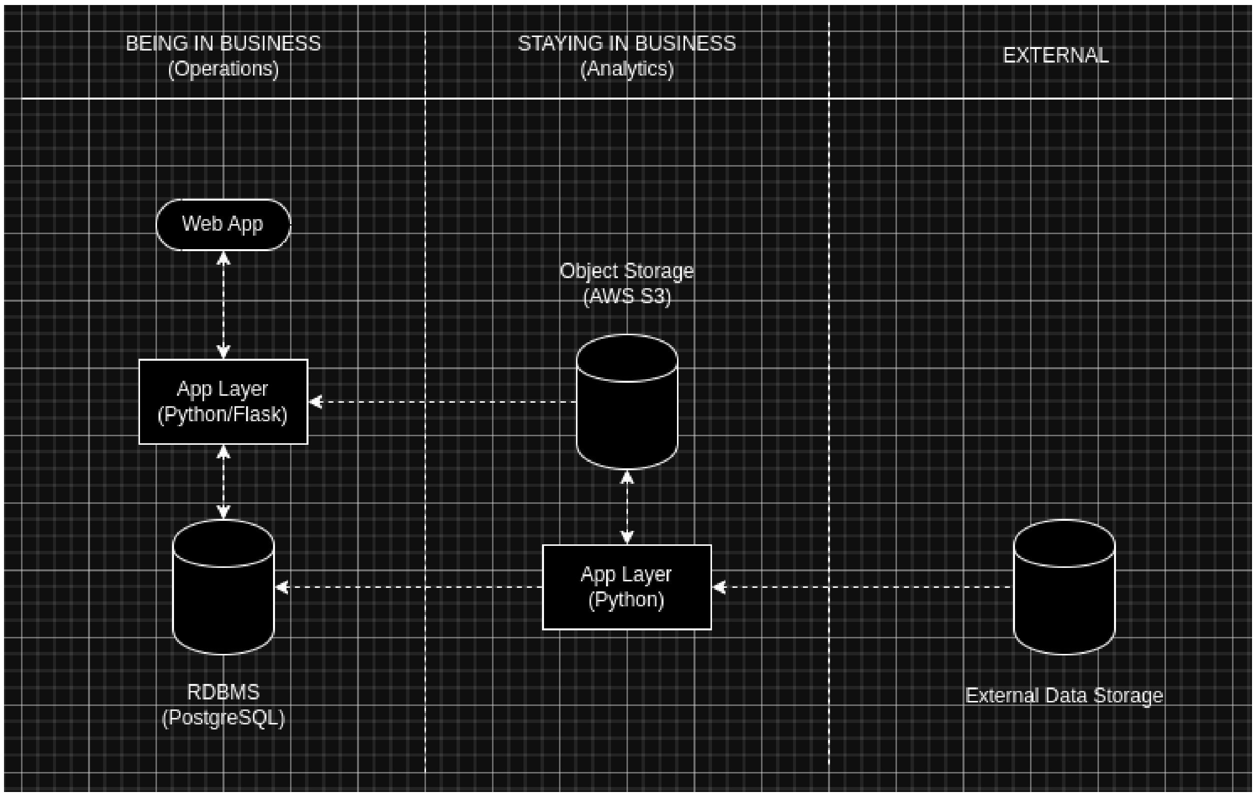
Type	Percentage
no_personal_information	20
personal_information_risky	15
personal_information_not_risky	10

If a higher percentage of the source data is risky we would proportionally increase the no_personal_information percentage each time we train the model. This would allow us to charge the right percentage as the risk in the population changes. We might not know the risk of a customer who doesn't give us personal information but based on the population we can make an informed guess where on average we are covering our warranty risk. We can also consider that as the percentage of non-risky people decreases, we can charge a bit less for no_personal_information and far less for personal_information_not_risky since we are only giving this greatly discounted price to very few warranties.

Elaborate on the reference architecture

I've included an updated version of the diagram from part 2 of the project below which shows the reference architecture for the entire application spanning the categories "Being in Business", "Staying in Business", and "External". While this architecture was built to support fetching some specific external data, training a model for that data, and making predictions for warranty risk, it easily scales to any type of external data and any type of model we could imagine.

To extend to a new external data set and new ML algorithm for training, we can simply create a new instance in the Analytics boundary for a new data type. Just like before, we can imagine a recurring job which takes the external data and creates a CSV file in S3 at some interval. When this is done we also have a job which trains the model using this newly created CSV file and saves this model to S3 as well. This latest model will be the one that is always retrieved from the operations side when needing to run some inference.



Note that the arrows are meant to show the direction that data flows in the reference architecture. Note that when adding a new external data source, we can easily create a new job in the App Layer on the Analytics side of the business. This App Layer now has access to the RDBMS to update percentages used to calculate warranty prices explained above depending on changes to the ingested, external personal information we use to train our models. In the example we gave for predicting wage changes for employees, the above change also means that we can write to the payroll table as well from the Analytics side. This will allow us to periodically update income levels for each employee.

We've shuffled the position of the App Layer and Object Storage in the Analytics side of the business to make the diagram a bit easier to read so it no longer has storage at the same level and each App Layer at the same level across all parts of the system. This doesn't have any impact on the design of the system, just on the diagram for illustrative purposes. All previous explanations about the different parts of the system from part 2 of the project still apply. Please consult part 2 of the project for additional information about the reference architecture.

Leverage cloud capabilities

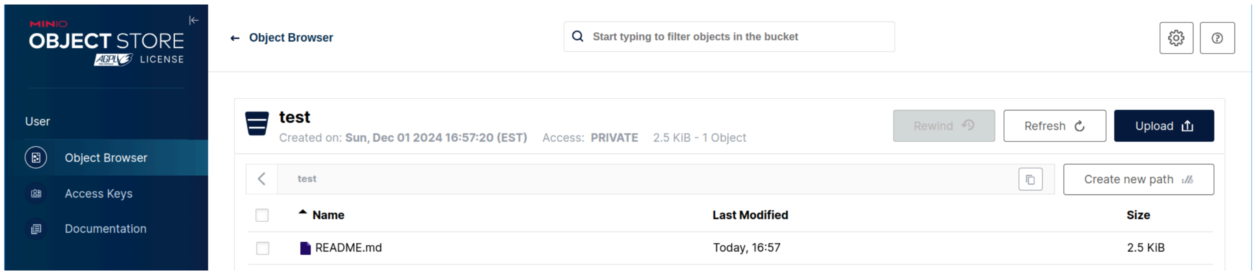
As I mentioned previously, I'm using Heroku as a cloud and managed service provider to build out the different parts of my application. In terms of big data capabilities, Heroku offers an RDBMS (Heroku PostgreSQL) and Stackhero S3 (Object Storage) through add-ons. We'll be leveraging both to implement the above reference architecture. Heroku offers NoSQL through various add-ons but since I'll be using CSV files directly I currently don't plan to use it.

Heroku PostgreSQL

I've already configured PostgreSQL both locally and remotely on Heroku. This was explained in the first section of this report. I'm using PostgreSQL currently for the main application database and won't be leveraging it for now as part of the Analytics side.

Stackhero S3 (Object Storage)

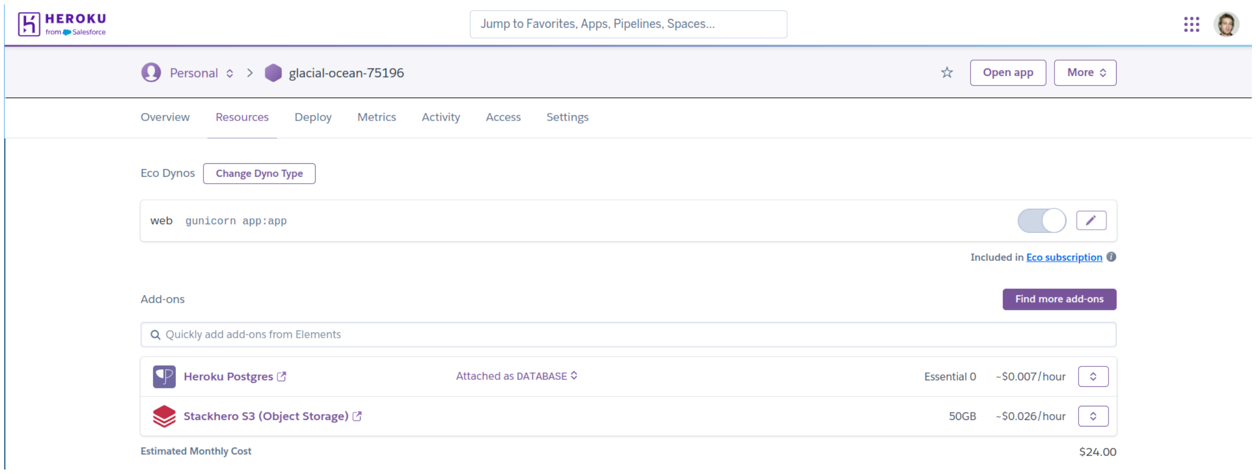
Stackhero is a separate service that allows easy configuration of Amazon S3. Previously, during the beginning of the semester I've set up AWS S3 manually but I've decided to go with Stackhero S3 for this project which makes it a little easier to setup and deal with in Heroku since it's already nicely integrated. I'm able to write and retrieve files from a test bucket that I've got working for this part of the project. I've included screenshots below which show a test file I've uploaded to the object storage and Stackhero S3 in the admin panel. In the final part, I'll be integrating this service with the App Layer on both sides.



test
Created on: Sun, Dec 01 2024 16:57:20 (EST) Access: PRIVATE 2.5 KiB - 1 Object

test

Name	Last Modified	Size
README.md	Today, 16:57	2.5 KiB



Eco Dynos

```
web   gunicorn app:app
```

Included in Eco subscription

Add-ons

Heroku Postgres	Attached as DATABASE	Essential 0	~$0.007/hour
Stackhero S3 (Object Storage)		50GB	~$0.026/hour

Estimated Monthly Cost $24.00

ObjectRocket for MongoDB

This is the MongoDB add-on provided by another third-party. I don't have a need for this at the moment due to my current architecture but if plans change this is what I'll be using.