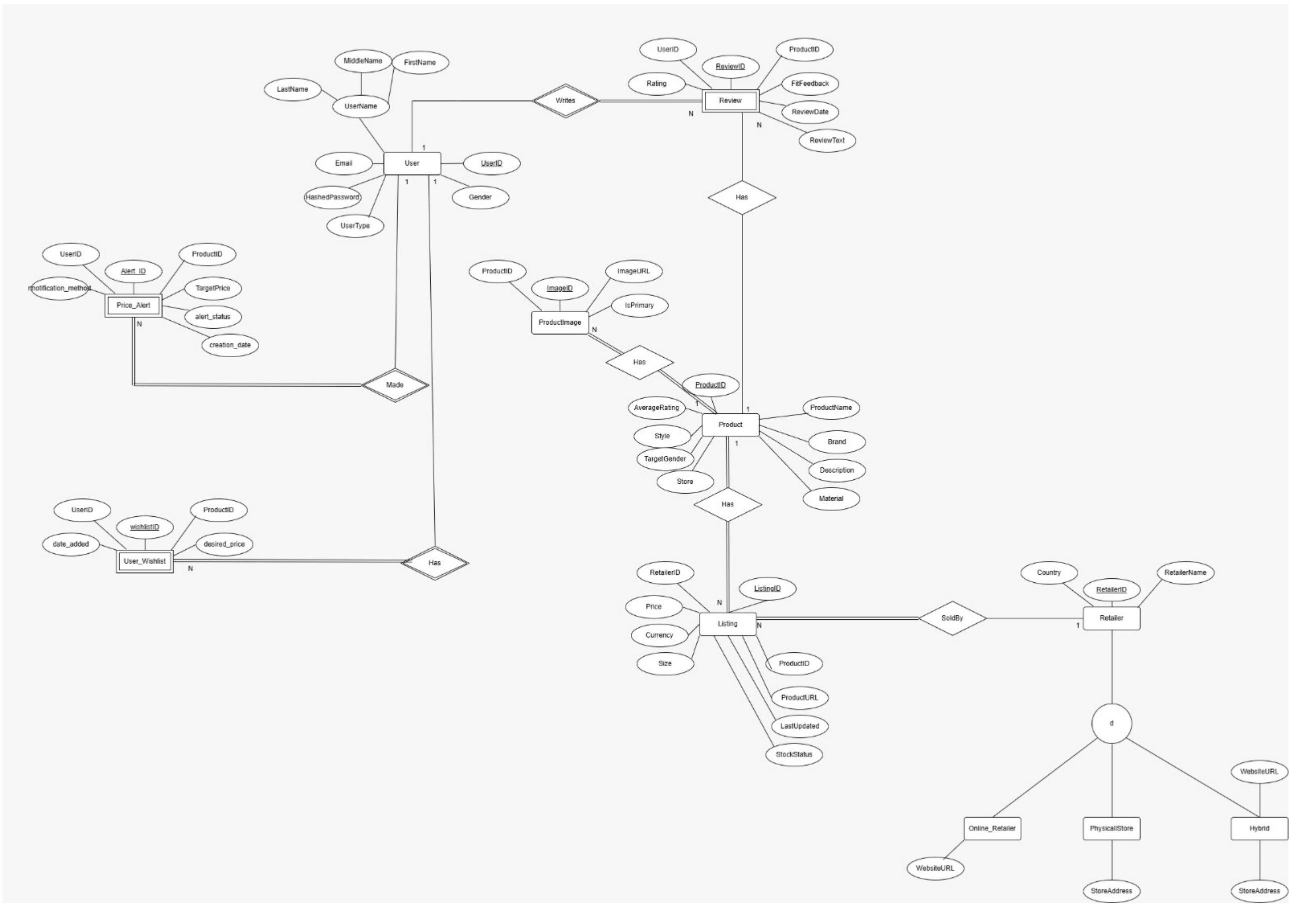
COS221 Assignment: SQL Legends

Task 2: First iteratiom

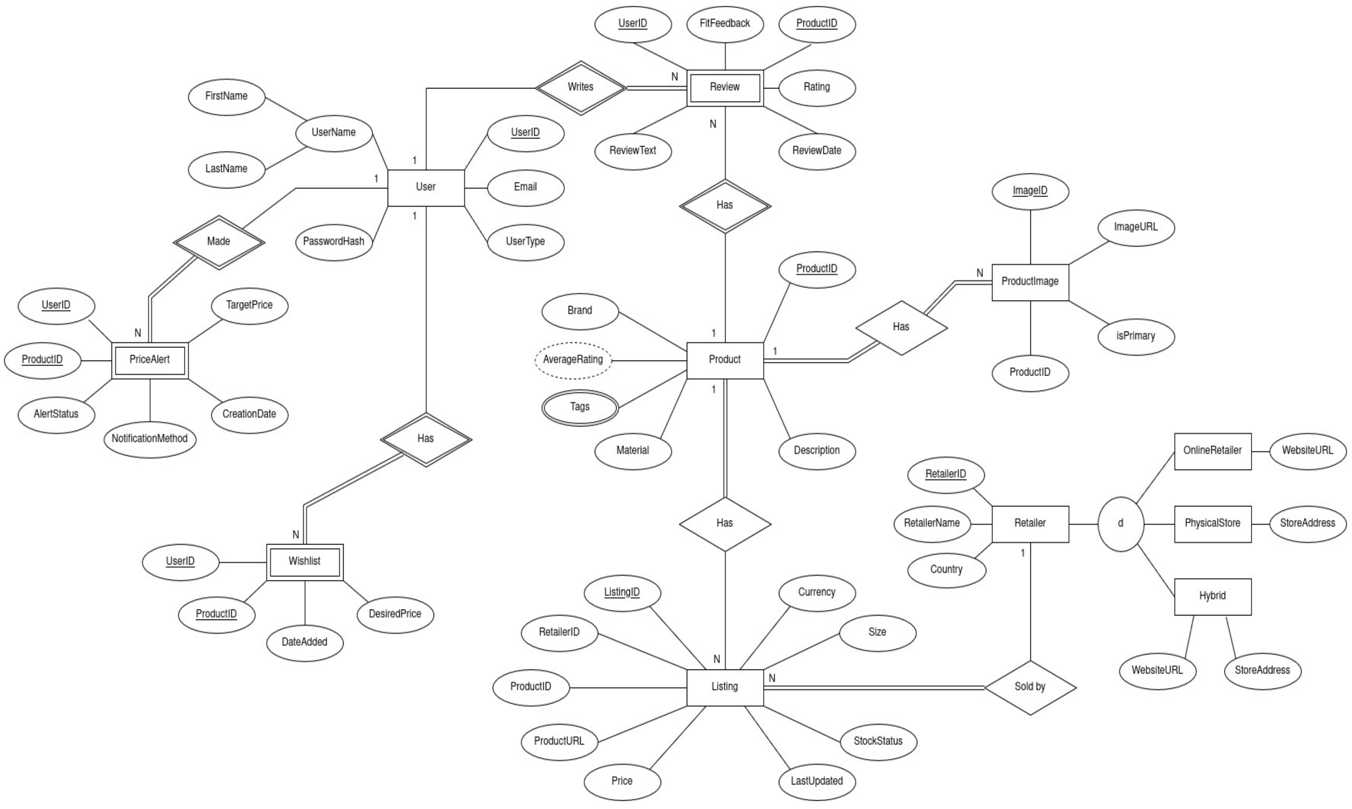


For this assignment, the assumption was made that one product could have multiple listings and a decision was made to represent this information with a one to many relationship, as opposed to a solution which would see identical products with separate listings represented as “two distinct products”.

Composite attribute username was later changed to not include a middle name, and store was removed from product, as we realised that it should rather be an attribute of the listing and not the product.

The final diagram shown below also removes gender from the user entity, as it was decided that products would not be filtered by gender. Average rating was found to be computable and therefore the final iteration shows it as a derived entity.

Task 2: Final:



In this models users are allowed to have multiple wishlist items, create multiple price alerts and write reviews for products.

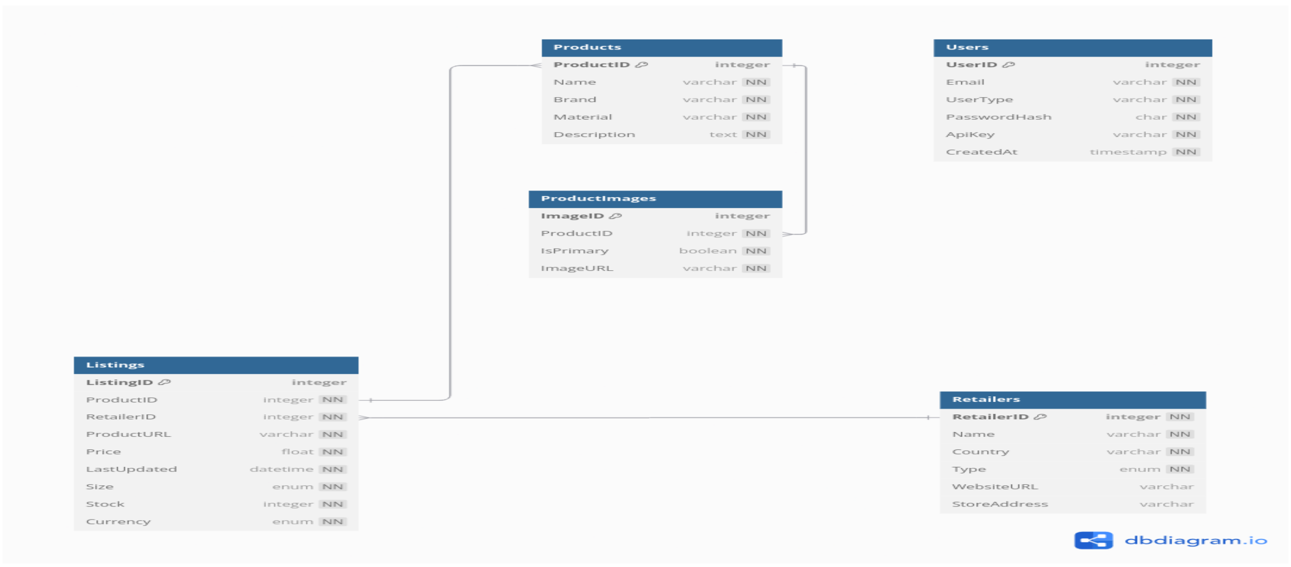
Products may have any number of reviews including 0, but only one review from each user, and must have at least one image. Listings must include a product and any number of listings can be created by a retailer.

Retailers are split up into one of three types of establishments depending on whether they are online stores, physical stores or a hybrid of the two.

Weak entities such as price alerts, wishlists and reviews are all easily identified by a combination of the user they were created by and a product they reference.

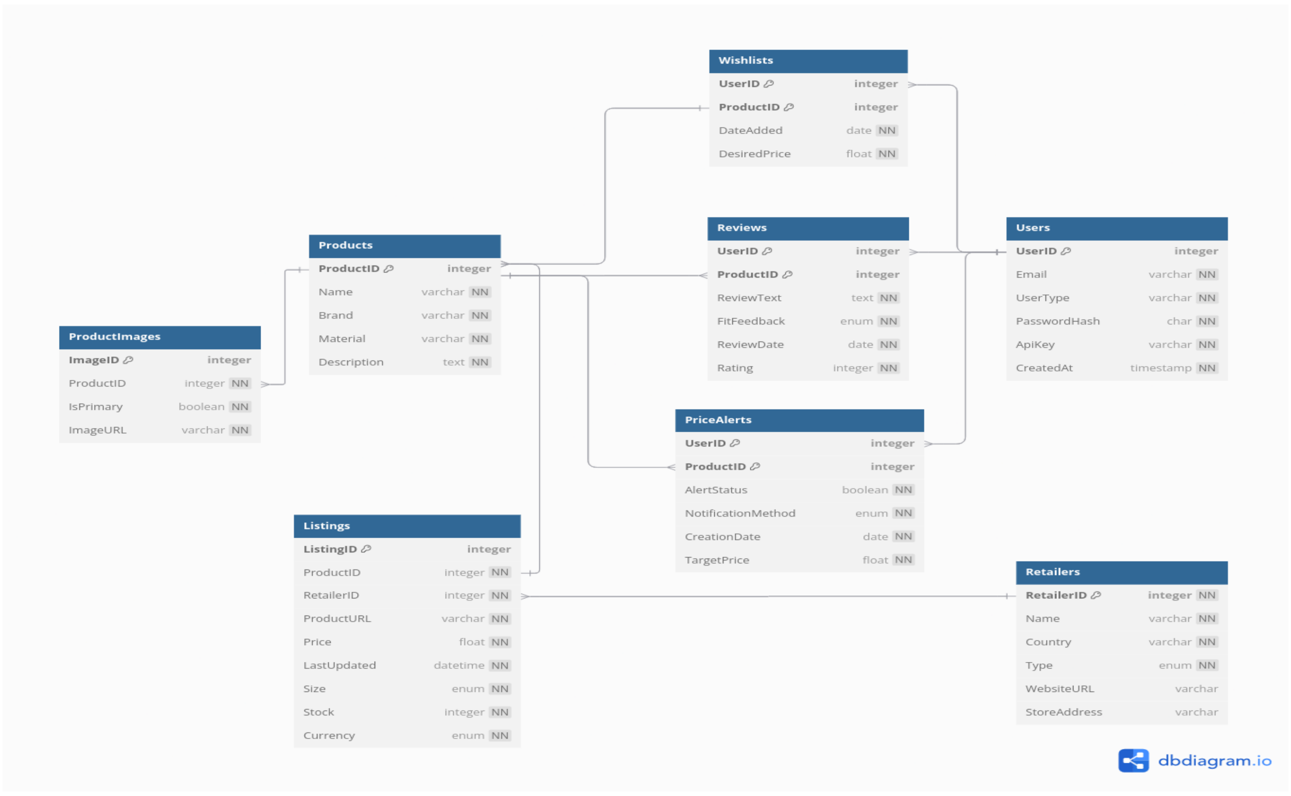
Task 3:

Step 1: Strong entities, namely products, listings, users, retailers and product image as well as their simple attributes are mapped using dbdiagram.io with results shown below:



Non nullable attributes are shown with symbol “NN” in the diagram and the datatype as well as primary key are indicated.

Step 2: Weak entity types are mapped along with their attributes as shown below:

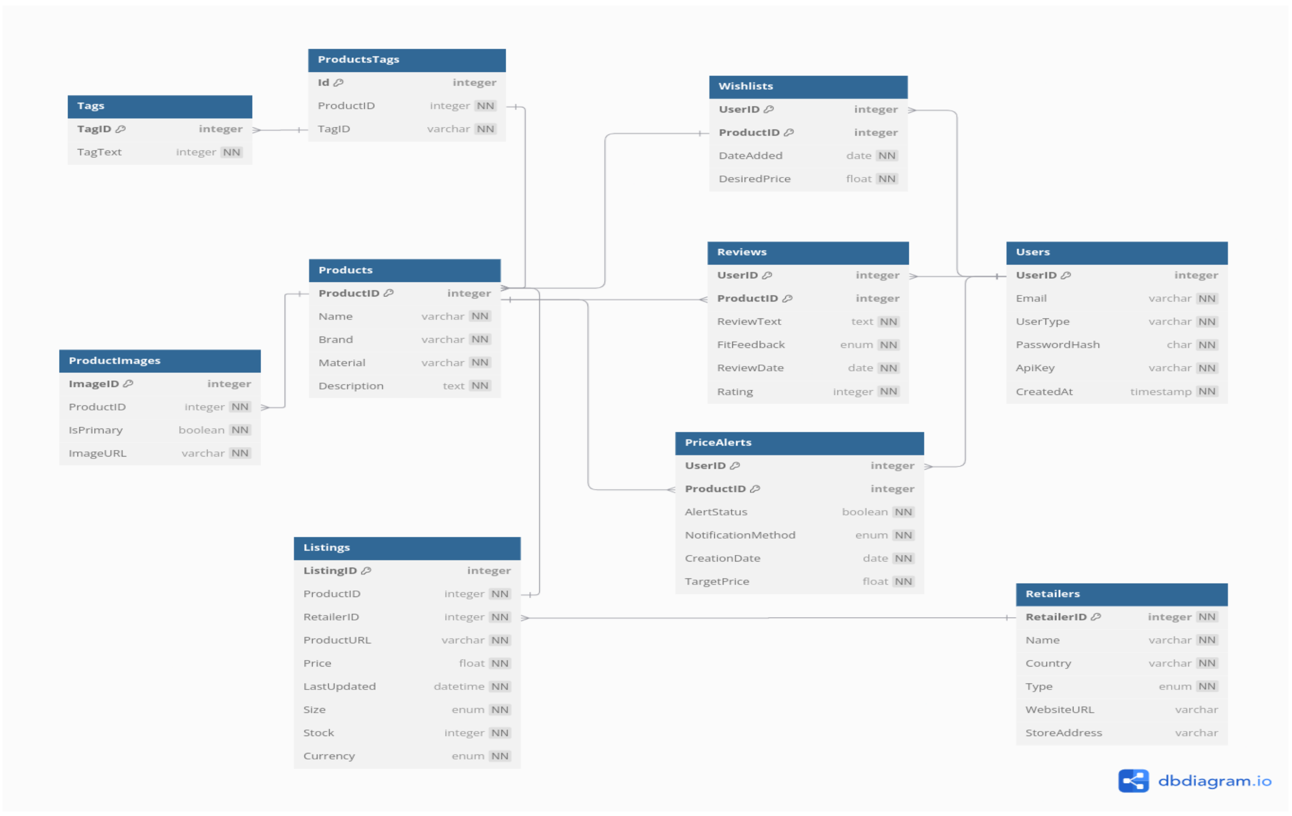


Weak entities receive primary keys based on their parent entities as prescribed and foreign keys are shown above.

Step 3: No one to one binary relationships are apparent and therefore we proceed to the next step

Step 4: Already shown above, the foreign key approach is used to ensure that there are no null values.

Step 5 and 6: Due to the fact that it was decided that the originally envisioned tags attribute (multivalued) would be better structured as a separate relation as suggested in step 6 of the lecture notes, to simplify search and filter functions whilst avoiding json array fields. An M:N relationship was created between products and tags. 2 new relations were created, one for the tags, and the next, a products tags relation to manage the M:N relationship successfully. The updated mapping is shown below:



Tags		
TagID	integer	
TagText	integer	NN

ProductsTags		
Id	integer	
ProductID	integer	NN
TagID	varchar	NN

Wishlists		
UserID	integer	
ProductID	integer	
DateAdded	date	NN
DesiredPrice	float	NN

Products		
ProductID	integer	
Name	varchar	NN
Brand	varchar	NN
Material	varchar	NN
Description	text	NN

Reviews		
UserID	integer	
ProductID	integer	
ReviewText	text	NN
FitFeedback	enum	NN
ReviewDate	date	NN
Rating	integer	NN

Users		
UserID	integer	
Email	varchar	NN
UserType	varchar	NN
PasswordHash	char	NN
ApiKey	varchar	NN
CreatedAt	timestamp	NN

ProductImages		
ImageID	integer	
ProductID	integer	NN
IsPrimary	boolean	NN
ImageURL	varchar	NN

PriceAlerts		
UserID	integer	
ProductID	integer	
AlertStatus	boolean	NN
NotificationMethod	enum	NN
CreationDate	date	NN
TargetPrice	float	NN

Listings		
ListingID	integer	
ProductID	integer	NN
RetailerID	integer	NN
ProductURL	varchar	NN
Price	float	NN
LastUpdated	datetime	NN
Size	enum	NN
Stock	integer	NN
Currency	enum	NN

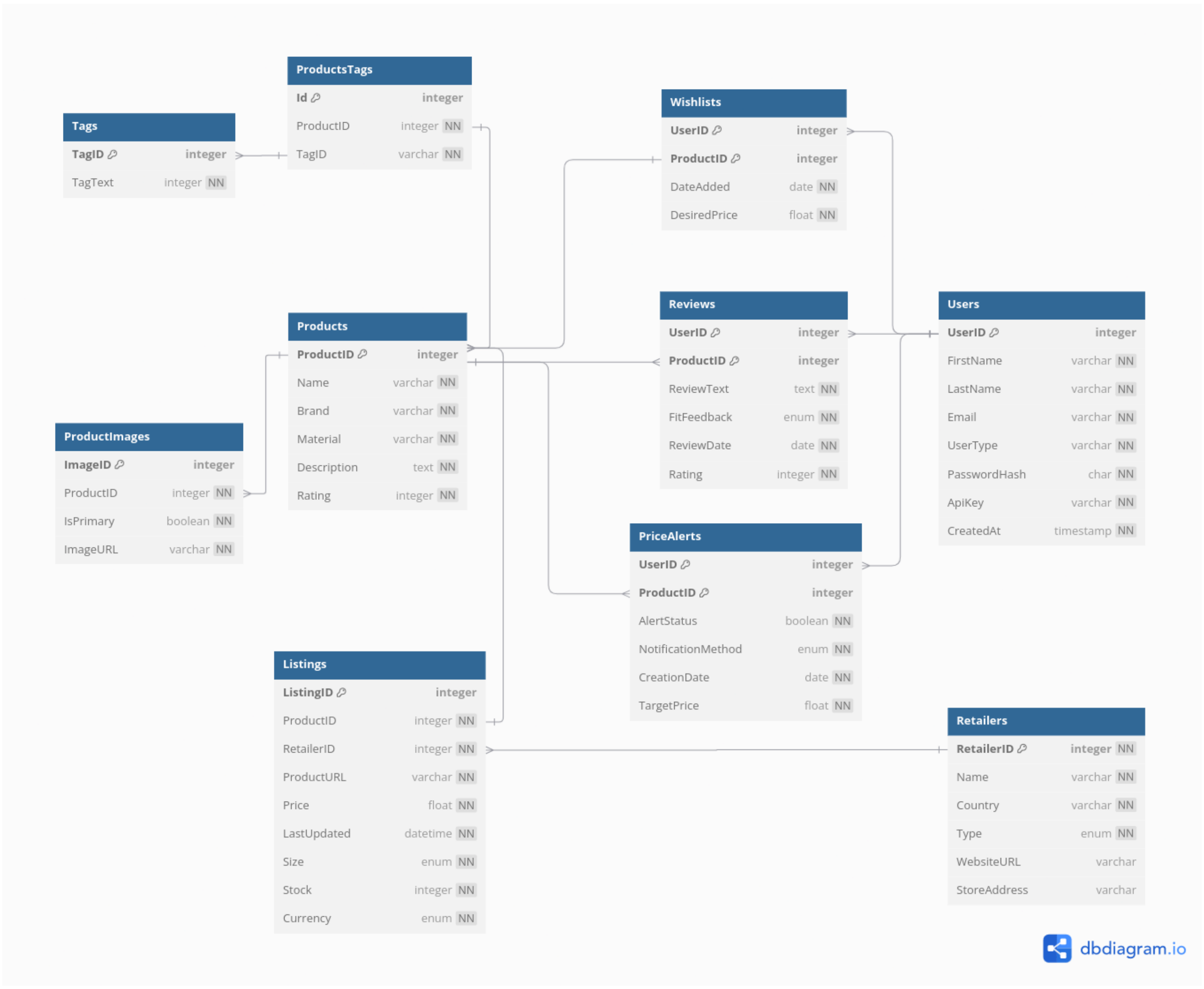
Retailers		
RetailerID	integer	NN
Name	varchar	NN
Country	varchar	NN
Type	enum	NN
WebsiteURL	varchar	
StoreAddress	varchar	

Step 7: No N-ary relationships are observed and therefore this step can be skipped.

Step 8: Retailer classes are collapsed into a singular type in accordance with method 8C and shown above in the retailer relation. Although this produces fields with potential null-values the advantage of having all of the information for a retailer in a single relation prompted this decision, and the limited attributes associated with each type mean that not a lot of space is used by this method. If many more attributes were associated with the subtypes, this method may have been unideal due to storage constraints.

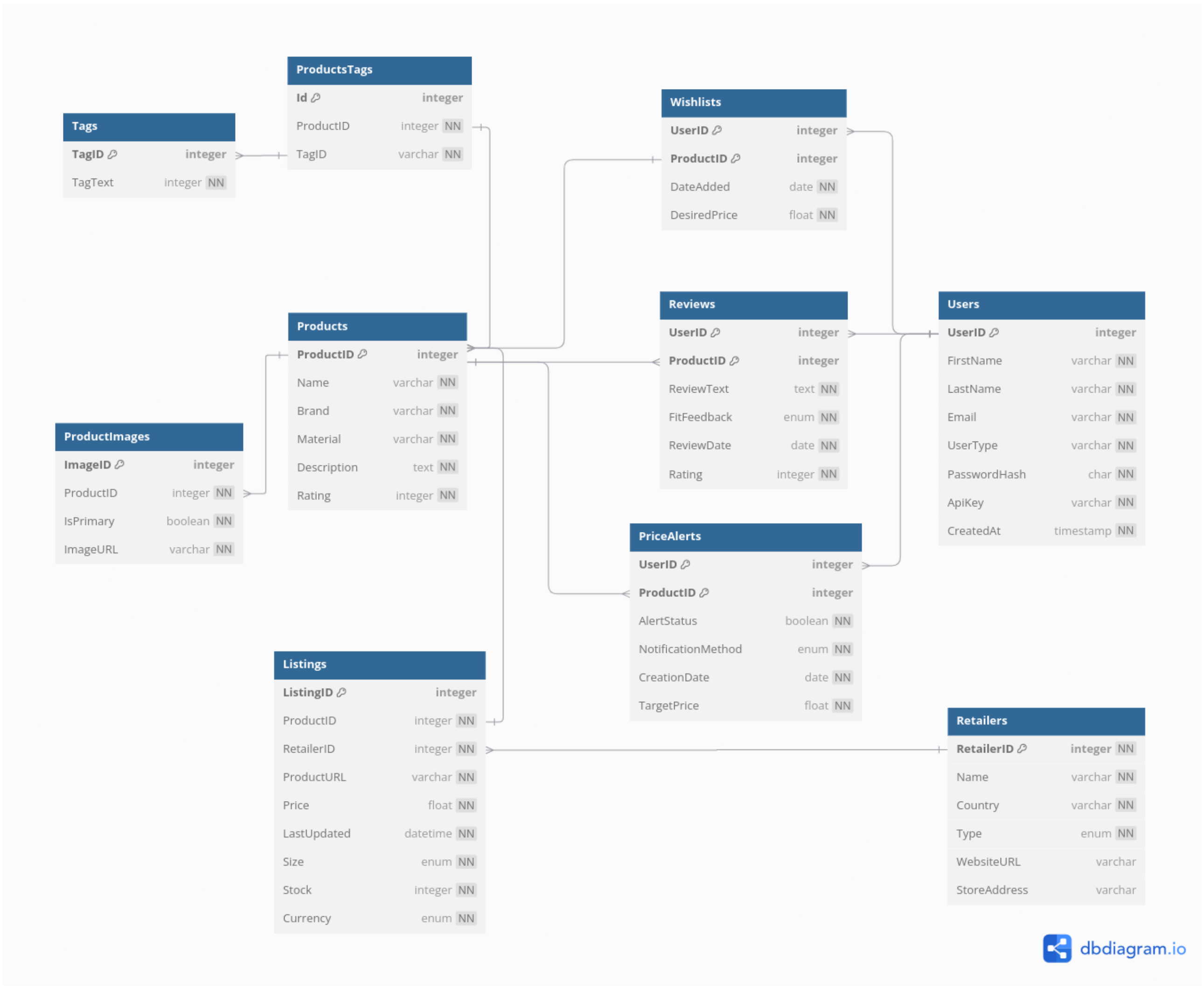
Step 9: No unions are observed and therefore we proceed to the final mapped diagram

Final Mapping



Please note that optional relationships can not always be shown clearly with the dbdiagram.io software. For the mapping all of the relationship constraints are kept the same as indicated in the original ER diagram. Username has been collapsed into its simple attributes and optional fields in relation Retailers are shown without the NN symbol indicating that they are nullable fields. Length constraints can also not be shown but are included in task 4 where the database is created.

Task 4:



The above summarised schema shown and explained in the mapping section is created using the following SQL statements:

```
-- Database: `u24742083_CompareIt`
--
CREATE DATABASE IF NOT EXISTS `u24742083_CompareIt` DEFAULT CHARACTER SET
utf8mb4 COLLATE utf8mb4_general_ci;
USE `u24742083_CompareIt`;
```

```sql
--
-- Table structure for table `Listings`
--

CREATE TABLE `Listings` (
  `ListingID` int(11) NOT NULL,
  `ProductID` int(11) NOT NULL,
  `RetailerID` int(11) NOT NULL,
  `ProductURL` varchar(255) NOT NULL,
  `Price` decimal(10,2) NOT NULL CHECK (`Price` >= 0),
  `LastUpdated` datetime NOT NULL,
  `Size` enum('S','M','L','XL') DEFAULT 'M',
  `Stock` enum('in_stock','out_of_stock') NOT NULL,
  `Currency` enum('USD','ZAR','EUR') NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;


--
-- Table structure for table `Products`
--

CREATE TABLE `Products` (
  `ProductID` int(11) NOT NULL,
  `Name` varchar(255) NOT NULL,
  `Brand` varchar(100) NOT NULL,
  `Material` varchar(100) NOT NULL,
  `Description` text NOT NULL,
  `Rating` int(11) NOT NULL CHECK (`Rating` between 1 and 5),
  `Color` varchar(50) NOT NULL DEFAULT 'Unknown'
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;


--
-- Table structure for table `ProductsTags`
--

CREATE TABLE `ProductsTags` (
  `Id` int(11) NOT NULL,
  `ProductID` int(11) NOT NULL,
  `TagID` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

```sql
--
-- Table structure for table `Retailers`
--

CREATE TABLE `Retailers` (
  `RetailerID` int(11) NOT NULL,
  `Name` varchar(100) NOT NULL,
  `Country` varchar(100) NOT NULL,
  `Type` enum('online','physical','hybrid') NOT NULL,
  `WebsiteURL` varchar(255) DEFAULT NULL,
  `StoreAddress` varchar(255) DEFAULT NULL,
  `IsActive` tinyint(1) NOT NULL DEFAULT 1,
  `ManagerUserID` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Table structure for table `Wishlists`
--

CREATE TABLE `Wishlists` (
  `UserID` int(11) NOT NULL,
  `ProductID` int(11) NOT NULL,
  `DateAdded` date NOT NULL,
  `DesiredPrice` decimal(10,2) NOT NULL CHECK (`DesiredPrice` >= 0)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Indexes and Constraints for all tables:

```
--
-- Indexes for table `Listings`
--
ALTER TABLE `Listings`
  ADD PRIMARY KEY (`ListingID`),
  ADD UNIQUE KEY `ProductID` (`ProductID`,`RetailerID`),
  ADD KEY `RetailerID` (`RetailerID`),
  ADD KEY `idx_product_retailer` (`ProductID`,`RetailerID`);

--
-- Indexes for table `PriceAlerts`
--
ALTER TABLE `PriceAlerts`
  ADD PRIMARY KEY (`UserID`,`ProductID`),
  ADD KEY `ProductID` (`ProductID`);

--
-- Indexes for table `ProductImages`
--
ALTER TABLE `ProductImages`
  ADD PRIMARY KEY (`ImageID`),
  ADD KEY `ProductID` (`ProductID`);

--
-- Indexes for table `Products`
--
ALTER TABLE `Products`
  ADD PRIMARY KEY (`ProductID`),
  ADD KEY `idx_rating` (`Rating`);

--
-- Indexes for table `ProductsTags`
--
ALTER TABLE `ProductsTags`
  ADD PRIMARY KEY (`Id`),
  ADD UNIQUE KEY `ProductID` (`ProductID`,`TagID`),
  ADD KEY `TagID` (`TagID`),
  ADD KEY `idx_product_tag` (`ProductID`,`TagID`);

--
-- Indexes for table `Retailers`
--
ALTER TABLE `Retailers`
  ADD PRIMARY KEY (`RetailerID`),
  ADD UNIQUE KEY `Name` (`Name`),
  ADD KEY `fk_manager_user` (`ManagerUserID`);
```

```sql
--
-- Indexes for table `Reviews`
--
ALTER TABLE `Reviews`
  ADD PRIMARY KEY (`UserID`,`ProductID`),
  ADD KEY `idx_product_user` (`ProductID`,`UserID`);

--
-- Indexes for table `Tags`
--
ALTER TABLE `Tags`
  ADD PRIMARY KEY (`TagID`),
  ADD UNIQUE KEY `TagText` (`TagText`);


--
-- Indexes for table `Users`
--
ALTER TABLE `Users`
  ADD PRIMARY KEY (`UserID`),
  ADD UNIQUE KEY `Email` (`Email`),
  ADD UNIQUE KEY `ApiKey` (`ApiKey`),
  ADD KEY `idx_email` (`Email`);

--
-- Indexes for table `Wishlists`
--
ALTER TABLE `Wishlists`
  ADD PRIMARY KEY (`UserID`,`ProductID`),
  ADD KEY `ProductID` (`ProductID`);

--
-- AUTO_INCREMENT for dumped tables
--

--
-- AUTO_INCREMENT for table `Listings`
--
ALTER TABLE `Listings`
  MODIFY `ListingID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=69877;

--
-- AUTO_INCREMENT for table `ProductImages`
--
ALTER TABLE `ProductImages`
  MODIFY `ImageID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=5313;
```

```sql
--
-- AUTO_INCREMENT for table `Products`
--
ALTER TABLE `Products`
  MODIFY `ProductID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=5622;

--
-- AUTO_INCREMENT for table `ProductsTags`
--
ALTER TABLE `ProductsTags`
  MODIFY `Id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=10625;

--
-- AUTO_INCREMENT for table `Retailers`
--
ALTER TABLE `Retailers`
  MODIFY `RetailerID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=51922;

--
-- AUTO_INCREMENT for table `Tags`
--
ALTER TABLE `Tags`
  MODIFY `TagID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=11;

--
-- AUTO_INCREMENT for table `Users`
--
ALTER TABLE `Users`
  MODIFY `UserID` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=24448;

--
-- Constraints for dumped tables
--

--
-- Constraints for table `Listings`
--
ALTER TABLE `Listings`
  ADD CONSTRAINT `Listings_ibfk_1` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE,
  ADD CONSTRAINT `Listings_ibfk_2` FOREIGN KEY (`RetailerID`) REFERENCES
`Retailers` (`RetailerID`);

--
-- Constraints for table `PriceAlerts`
--
ALTER TABLE `PriceAlerts`
  ADD CONSTRAINT `PriceAlerts_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `Users`
(`UserID`) ON DELETE CASCADE,
  ADD CONSTRAINT `PriceAlerts_ibfk_2` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE;
```

```sql
--
-- Constraints for table `ProductImages`
--
ALTER TABLE `ProductImages`
  ADD CONSTRAINT `ProductImages_ibfk_1` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE;

--
-- Constraints for table `ProductsTags`
--
ALTER TABLE `ProductsTags`
  ADD CONSTRAINT `ProductsTags_ibfk_1` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE,
  ADD CONSTRAINT `ProductsTags_ibfk_2` FOREIGN KEY (`TagID`) REFERENCES `Tags`
(`TagID`);

--
-- Constraints for table `Retailers`
--
ALTER TABLE `Retailers`
  ADD CONSTRAINT `fk_manager_user` FOREIGN KEY (`ManagerUserID`) REFERENCES
`Users` (`UserID`) ON DELETE SET NULL ON UPDATE CASCADE;

--
-- Constraints for table `Reviews`
--
ALTER TABLE `Reviews`
  ADD CONSTRAINT `Reviews_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `Users`
(`UserID`) ON DELETE CASCADE,
  ADD CONSTRAINT `Reviews_ibfk_2` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE;


--
-- Constraints for table `Wishlists`
--
ALTER TABLE `Wishlists`
  ADD CONSTRAINT `Wishlists_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `Users`
(`UserID`) ON DELETE CASCADE,
  ADD CONSTRAINT `Wishlists_ibfk_2` FOREIGN KEY (`ProductID`) REFERENCES
`Products` (`ProductID`) ON DELETE CASCADE;
COMMIT;
```

Task 6:

To populate our database with a rich and realistic set of data suitable for testing and demonstration, our team developed a custom Python script using the Faker library, CSV parsing, and structured data modeling. The goal was to simulate a full-scale e-commerce product catalog, complete with extensive product details, associated brands, multiple retailers (stockists), and varied user reviews.

We began by generating 850 unique products and 50 distinct brands, each with its own origin and name to ensure variety. To anchor our data in realism, we parsed a provided style dataset (from styles.txt), selectively extracting fields like gender, category, article type, base colour, season, and product display names. This metadata ensured that a portion of our products were grounded in real-world fashion attributes. For the remaining products, we algorithmically generated names, styles, categories, and associated metadata to extend variety beyond the static dataset.

Each product includes several layers of associated data:

Style tags were randomly sampled from five major fashion archetypes (casual, formal, sporty, trendy, vintage). These tags informed the selection of materials, which were mapped per style to maintain internal consistency.

Seasonality was either directly derived from the style data or logically grouped (e.g., spring/summer or autumn/winter) to reflect realistic seasonal wearability.

We included sustainability tags—such as "organic cotton" or "recycled polyester"—based on material choice, helping to simulate modern ethical branding considerations.

Stockists were programmatically generated for each product (10–15 per item), each with unique store names, dynamic pricing, inventory levels, and a mock store manager with full contact information. This was designed to mirror the structure of real retail platforms with multiple points of sale.

User reviews were added to every product, with each review including a user name, email, rating (1–5 stars), and a date. To reflect sentiment variance, reviews with lower ratings pulled from a curated list of negative feedback, while higher-rated reviews were selected from a positive comment pool. This dichotomy was intentional to support frontend testing for ratings breakdowns and review sorting features.

We also processed a secondary file (images.txt) to assign realistic product images. If image data was missing or malformed, fallback handling ensured the integrity of the product entry remained intact.

By writing all data to a single JSON structure (productsApi.json), we ensured compatibility with common API mocking or frontend testing frameworks. Our approach balanced authenticity with scale, using real data where available and supplementing it with intelligently generated content. This method allowed us to create a database that was not only comprehensive and realistic but also robust enough to test edge cases, UI behavior, and scalability.

Through this strategy, we ensured our data met the assignment's criteria for realism, completeness, diversity, and functionality—key components for both back-end reliability and front-end usability.

Task 7:

Indices were created for all fields to optimise the performance when searching in the database. The group discussed potentially only returning a limited number of products, but this was decided against to allow for smooth browsing after the initial loading of products.

The average of the reviews for each product, which was originally calculated at runtime each time a product was loaded was changed within the database to be a stored field for performance reasons, as due to the fact that very many products need to be loaded and thousands of reviews may exist per product, calculating this field causes performance issues.

Now when a new review is created, the relevant field in the Product table is updated. This has greatly increased performance due to the lower amount of queries executed when products are loaded.

Bonus features:

Secure login with re-captcha was implemented to handle malicious login attempts. Api keys stored in session storage are encrypted . Passwords are stored in hashed form, hashed by using the Argon2id algorithm as well as a salt to ensure secure storage.

Link to github repository:

https://github.com/logi4ace-coder/COS221-Assignment