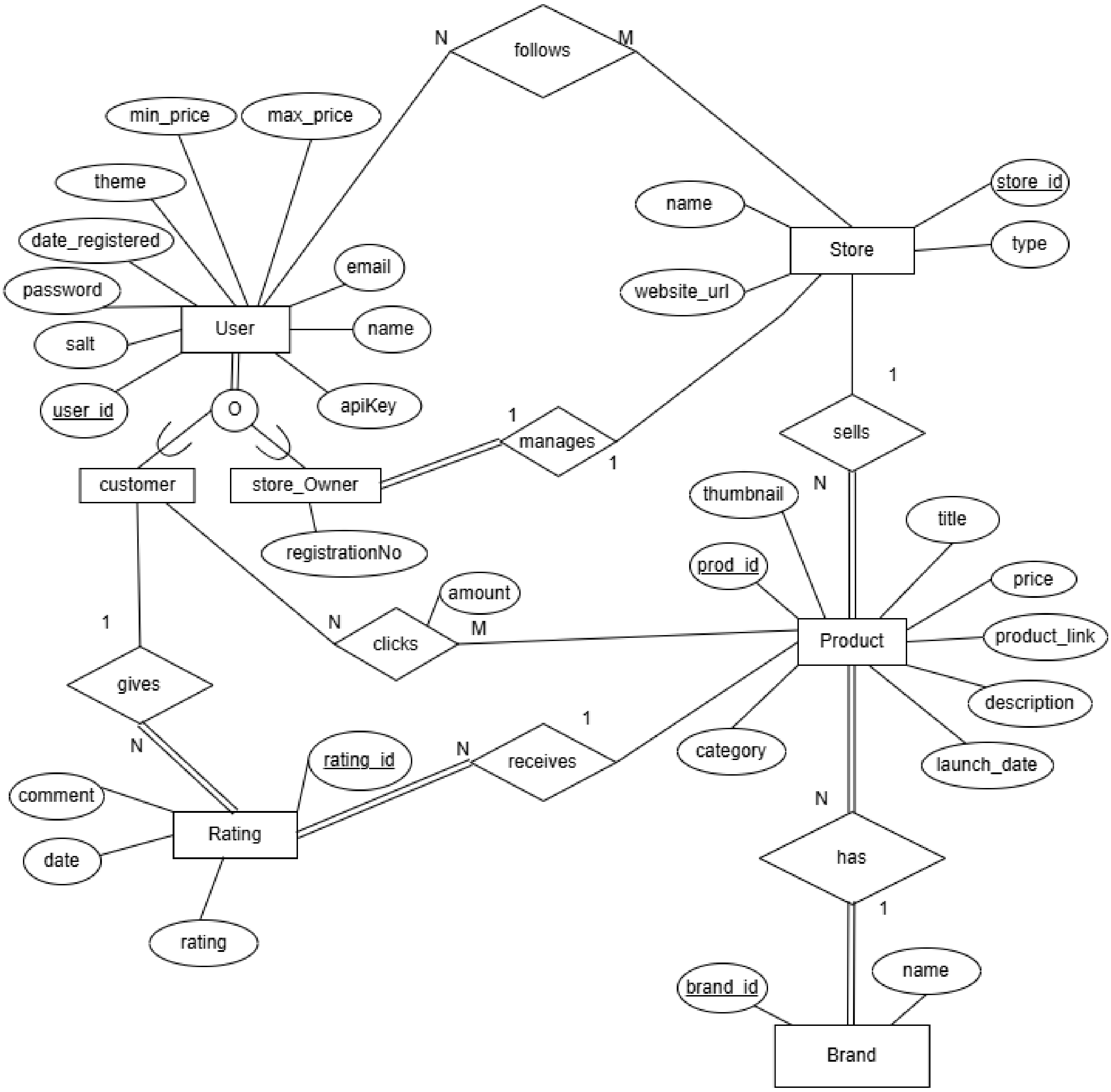
Database Docs

EER Diagram



EER to Relational Mapping

Step 1:

User

user_id	name	password	salt	email	theme	min_price	max_price	apiKey	date_registered

Product

prod_id	title	thumbnail	launch_date	product_link	price	description	category

Rating

rating_id	rating	date	comment

Store

store_id	name	type	url

Brand

brand_id	name

Step 2:

N/A

Step 3:

User

user_id	name	password	salt	email	theme	min_price	max_price	apiKey	date_registered

Product

prod_id	title	thumbnail	launch_date	product_link	price	description	category

Rating

rating_id	rating	date	comment

Store

store_id	name	type	url

Brand

brand_id	name

Step 4:

User

user_id	name	password	salt	email	theme	min_price	max_price	apiKey	date_registered

Brand

brand_id	name

Product

prod_id	title	thumbnail	launch_date	product_link	price	description	category	store_id	brand_id

Rating

rating_id	rating	date	comment	prod_id	user_id

Store

store_id	name	type	url

Step 5:

follows

store_id	user_id

User

user_id	name	password	salt	email	theme	min_price	max_price	apiKey	date_registered

Brand

brand_id	name

clicks

amount	prod_id	user_id

Product

prod_id	title	thumbnail	launch_date	product_link	price	description	category	store_id	brand_id

Rating

rating_id	rating	date	comment	prod_id	user_id

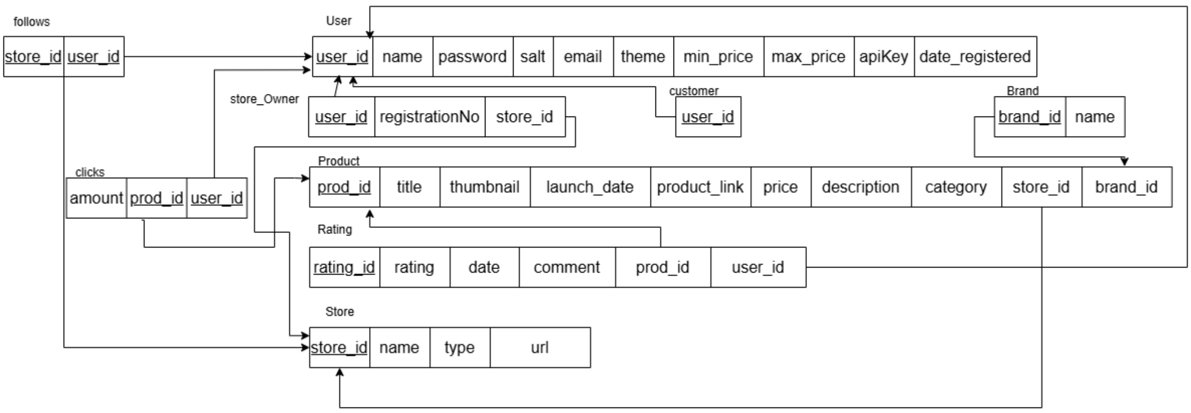
Store

store_id	name	type	url

Step 6: NOT APPLICABLE

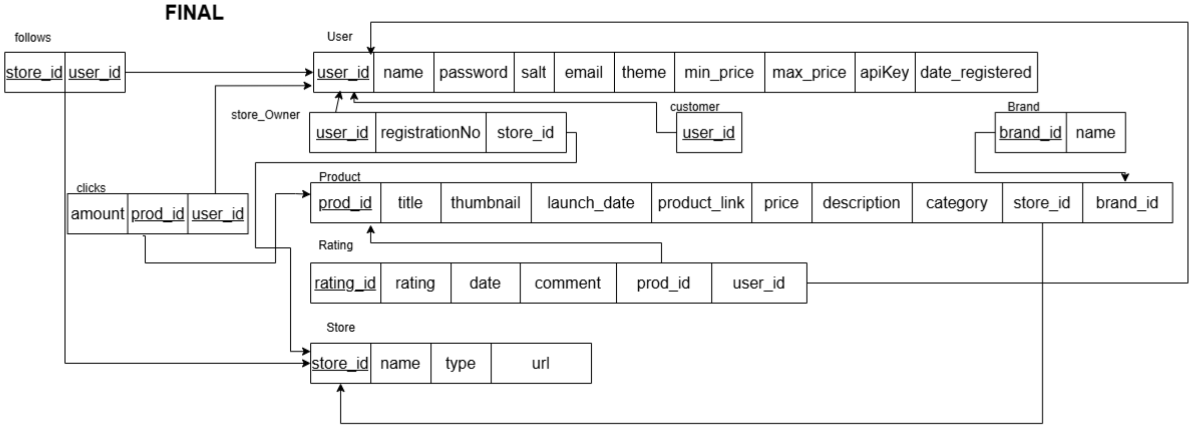
Step 7: NOT APPLICABLE

Step 8:

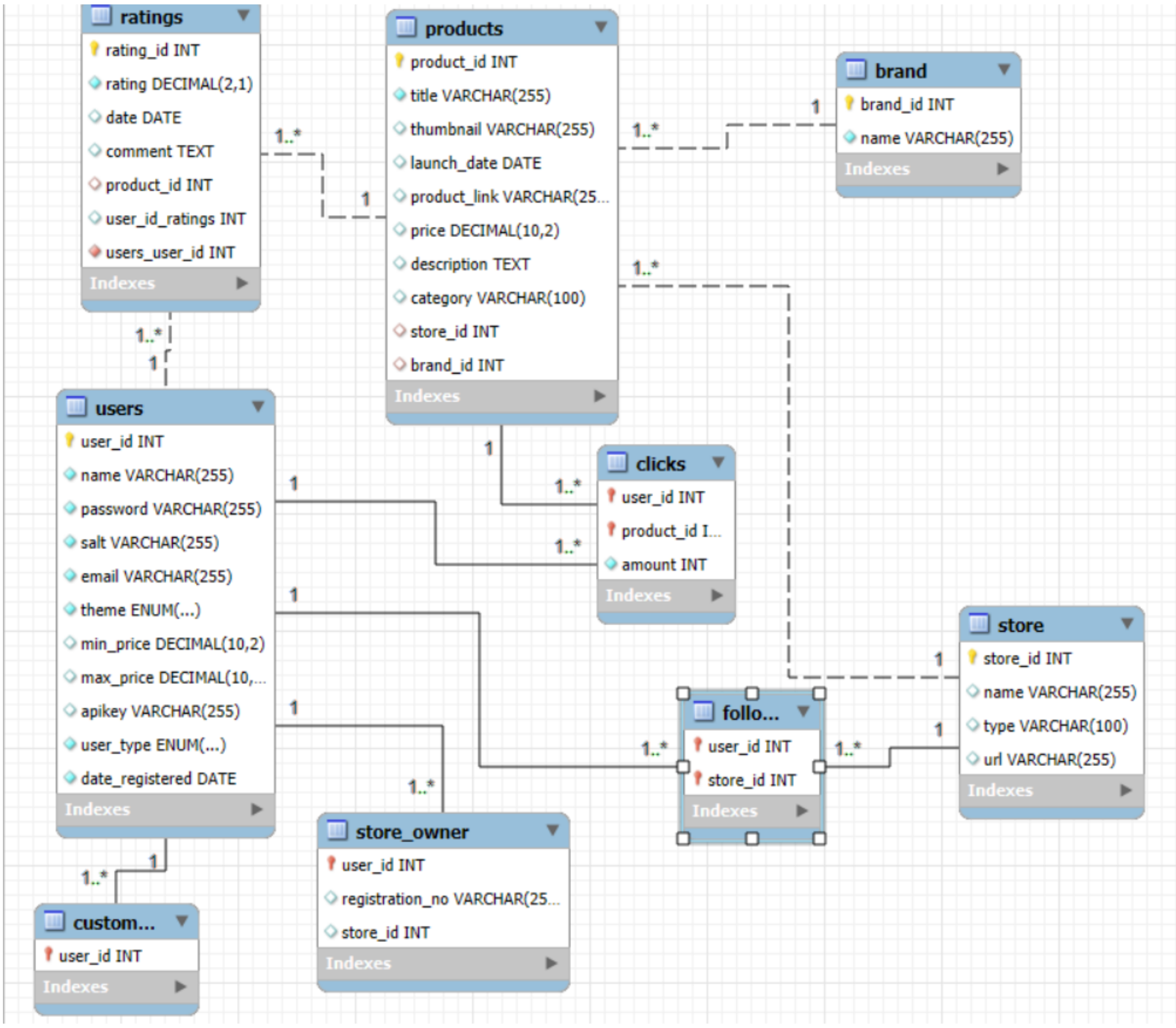


Step 9: NOT APPLICABLE

FINAL



Relational Schema



Data Population Method

We mainly used serpapi to get product data. We kept the data to South Africa to keep the currency consistent for South African stores.

Here is an example of a get request in the serpapi playground using google shopping:

https://serpapi.com/playground?engine=google_shopping&q=stationary
&location=South+Africa&google_domain=google.co.za&gl=za&hl=en

This returns a JSON object with a lot of parameters. The usable parameters are under shopping_results.

The parameters we will be retrieving is title, extracted_price, product_link, thumbnail and source.

The data is all stored in a JSON file according to the search term used to get the data and then we used a javascript file to retrieve the data and make an api call to insert the data into the database. For the category attribute we set the category to the search term used to retrieve the data from the api.

Before adding the products though we used the source to generate all the stores needed in the database and used ChatGPT to add links to all the stores and set the store types in the database.

When adding the products we then link the source attribute by using the store_id in the products table.

All other data items were made by using the website as we were testing.

Query Optimisation

The query we will be optimizing is the following:

SELECT p.*, AVG(r.rating) as average_rating

FROM products p

LEFT JOIN ratings r ON p.product_id = r.product_id

GROUP BY p.product_id ORDER BY average_rating DESC

Currently it takes 0.078 sec to retrieve the data:

In relational algebra the query can be written as follows:

$$\pi_{\text{(p.product\_id, title, thumbnail, launch\_date, product\_link, price, description, category, store\_id, brand\_id, average\_rating)}} (\sigma_{\text{(p.product\_id = r.product\_id OR r.product\_id = NULL}} (\text{products} \times ({}_{\text{product\_id}}\mathcal{F}_{\text{(AVERAGE(rating) AS average\_rating)}}(\text{ratings}))))$$

After optimising the relational algebra, we get:

$$\pi_{\text{(p.product\_id, title, thumbnail, launch\_date, product\_link, price, description, category, store\_id, brand\_id, average\_rating)}} (\text{products} \bowtie_{\text{product\_id=prod\_id}} \rho_{\text{(average\_rating, prod\_id)}} (\pi_{\text{(AVERAGE(rating), product\_id)}}({}_{\text{product\_id}}\mathcal{F}_{\text{(AVERAGE(rating))}}(\text{ratings}))))$$

Implementing this into SQL yields the following:

SELECT p.*, average_rating

FROM products AS p

LEFT JOIN (

SELECT AVG(r.rating) AS average_rating, product_id FROM ratings AS r GROUP BY r.product_id

) AS r ON p.product_id = r.product_id

ORDER BY average_rating DESC;

This now takes 0.016 sec to retrieve the data

This means that we have speed up the retrieval process by about 5 times!

This is because we are doing the aggregation first and then joining the tables.