CMPUT 410 Project Report

Group 6 - Black Market

Mikus Lorence, Mihai Oprescu, Michael Bessette, Mark Tupala

1 Basics

The Black Market is a collection of PHP pages with RESTful URLs and custom internal web services accessed via AJAX in order to provide users with constantly updating information directly from the MySQL backend. JQuery and minor JavaScript libraries are used for various client-side animations and effects, using CURL to communicate with partner marketplaces by returning JSON encoded strings. This was accomplished without the use of a framework.

2 Database Structure

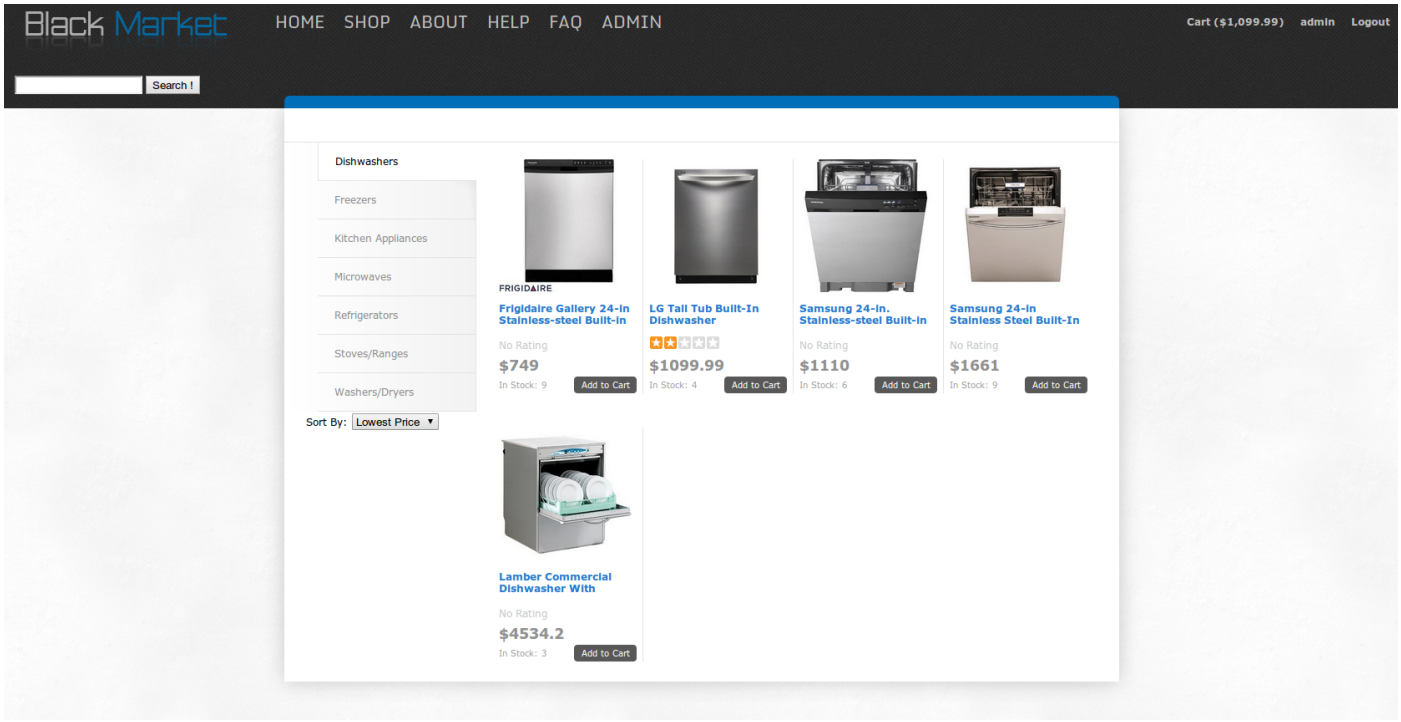
The Black Market's MySQL database is divided into nine tables. The *category* and *admin* tables list menu options for the shop menu and the management module, respectively. As expected *product* contains our entire inventory, and *user* stores all currently registered users, their personal information, as well as items in their cart (discussed below). Other tables that require product or user IDs use these tables as their foreign key reference. The *ratings* table stores the single rating that any customer can have for any one product. These values are aggregated incrementally and stored in the *product* table for ease of access. The remaining four tables store information relating to purchases. The central table is *userOrders*, which stores the user who made the purchase and the date on which it was delivered. Two other tables, *productOrders* (storing lists of products for each order) and *orderSources* (storing identical information for purchases made indirectly through the web service) reference the order ID in *userOrders* as their foreign key. The final table, *pendingOrders*, which does not reference *userOrders*, but rather stores a JSON encoded string representing a transaction that a user has not yet paid for. This information is removed and filtered into the appropriate tables upon receipt of payment.

3 Functionality

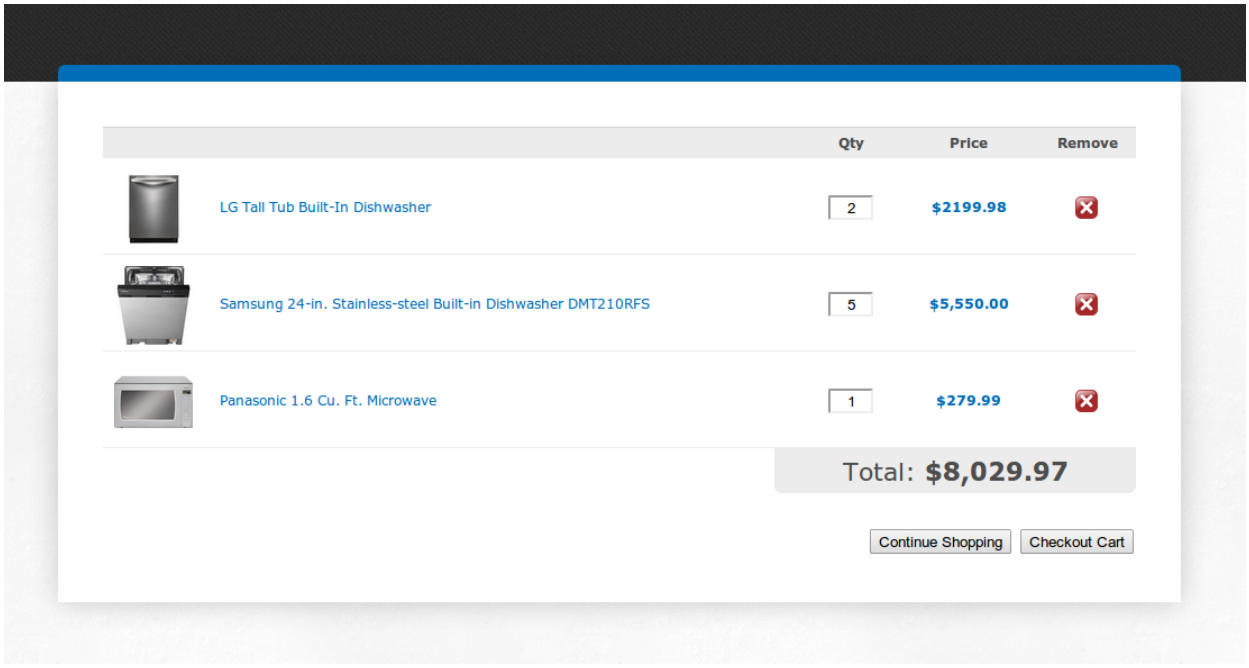
3.1 Modules

3.11 Purchase Module

The user is presented with this screen when they enter the shop page and click a category:



When a user visits their cart, they are presented with this page:



3.11.1 Shop page

Products Categories

While on the Shop page or any of the following sub-pages, a listing of all the categories is present on the left side. The products in each category can be sorted by the lowest or the highest price. The product view page displays more details about a product, and also allows you to rate the product if you purchased it.

Recommender System

A recommender system which produces the top (most sales) products is used on the front page. It is also used in the detailed view of a product in order to suggest to the user products that might be similar to their taste.

3.11.2 Ordering System

From the cart page, the user can choose to checkout their cart, which calls a few AJAX calls to the various internal service APIs in order to verify that the user is eligible to checkout their cart. The user is eligible if the following conditions are met:

- user is loggedin
    - a call is made to the authenticationServiceAPI with the parameter type = “checkLogin”
    - if user is not logged in, the login form sildes out and user is prompted
- the shopping cart is not empty
    - a call is made to cartService API with paramter type=”checkCart”

- if the cart is empty, a notice message animates on the page

- the user's shipping information is complete
    - a call is made to the authenticationService API again with parameter type = "checkAddress"
    - if the address is not filledo out, the user is prompted the with a message

Once it checks that these contraints are met, the user is taken to an order page where they are presented with a compact list of their cart and are given a chance to modify their cart and their shipping address. Upon verifying and clicking the "Place Order" button, a call is executed to the orderService API.

orderService API

```
POST /functionsPHP/orderService
params: none
return = {
        - relative URL back to the cart with an error param (in case of insufficient stock)
        - absolute URL to paybuddy
}
```

This service begins by going through a list of the products in the cart, and checking if they are in stock in our own store. If they are, it adds them to a temporary JSON object and goes on to the next.

If they are not, it grabs a list of all stores in the market, and records how much quantity each store has of that object. If the total quantity across all stores is insufficient, the service activates an "orderFailed" boolean, and modifies the cart to remove the insufficient quantities. It then goes on to process the rest of the cart. If the order was failed, it then takes the user to a page with their modified cart and informs them of the change.

If there is sufficient quantity across the different stores, the service attempts to rank them by ascending price, and goes through each store and adds them to the list of stores to purchase from until the needed quantitity is satisfied.

In the end, it compiles this JSON object which details exactly where to purchase each product and how much the store is charging for it:

```
{
   "total": 5407.98,
   "products": [
      {
         "pid": "c000032",
         "sources": [
            {
```

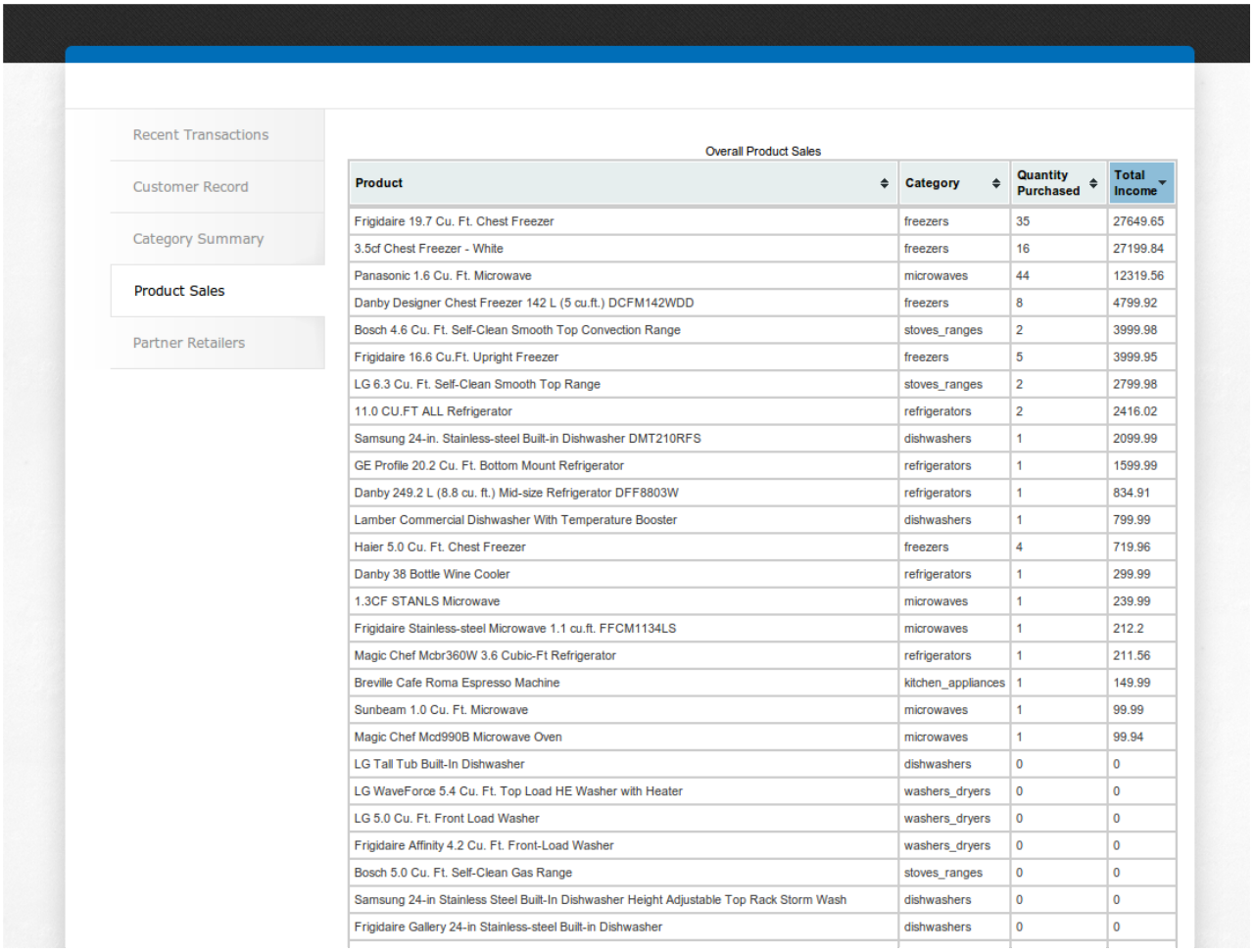
```
                "url": "home",
                "quantity": 1,
                "price": 1208.01,
                "name": "The Black Market"
            }
        ]
    },
    {
        "pid": "c000002",
        "sources": [
            {
                "url": "home",
                "quantity": 1,
                "price": 1399.99,
                "name": "The Black Market"
            },
             {
                "url": "home",
                "quantity": 2,
                "price": 1399.99,
                "name": "eMCS Market"
            }

        ]
    }
  ]
}
```


This object gets saved to a pendingOrders table in the database while the user processes their payment. When the user returns to the "receipt" page, a function grabs this object from the database, executes all the orders to other stores and subtracts quantities from our store. The pending order is then removed, and is logged in 3 other tables that detail who placed the order, what was on the order, and which external stores were bought from.

3.12 Management

The management module, or admin page, for the Black Market allows the logged in administrator to view statistics about the transactions taking place on the site. It is divided into five sections (History, Customers, Categories, Products, and Partners) that generate statistics at various depths. The tables displaying this information are loaded from server via an internal service (*adminService.php*) that generates the table with pre-defined queries specific to each granularity. These queries, in general, extract the list of "items of interest" (which define each depth) and run smaller queries to gather the remaining required data. Each table is accompanied by two text inputs that open date picking widgets to refine the time

frame used to generate the resulting tables. AJAX calls *adminService.php* to update the displayed information when the widgets close.



Overall Product Sales

Product	Category	Quantity Purchased	Total Income
Frigidaire 19.7 Cu. Ft. Chest Freezer	freezers	35	27649.65
3.5cf Chest Freezer - White	freezers	16	27199.84
Panasonic 1.6 Cu. Ft. Microwave	microwaves	44	12319.56
Danby Designer Chest Freezer 142 L (5 cu.ft.) DCFM142WDD	freezers	8	4799.92
Bosch 4.6 Cu. Ft. Self-Clean Smooth Top Convection Range	stoves_ranges	2	3999.98
Frigidaire 16.6 Cu.Ft. Upright Freezer	freezers	5	3999.95
LG 6.3 Cu. Ft. Self-Clean Smooth Top Range	stoves_ranges	2	2799.98
11.0 CU.FT ALL Refrigerator	refrigerators	2	2416.02
Samsung 24-in. Stainless-steel Built-in Dishwasher DMT210RFS	dishwashers	1	2099.99
GE Profile 20.2 Cu. Ft. Bottom Mount Refrigerator	refrigerators	1	1599.99
Danby 249.2 L (8.8 cu. ft.) Mid-size Refrigerator DFF8803W	refrigerators	1	834.91
Lamber Commercial Dishwasher With Temperature Booster	dishwashers	1	799.99
Haier 5.0 Cu. Ft. Chest Freezer	freezers	4	719.96
Danby 38 Bottle Wine Cooler	refrigerators	1	299.99
1.3CF STANLS Microwave	microwaves	1	239.99
Frigidaire Stainless-steel Microwave 1.1 cu.ft. FFCM1134LS	microwaves	1	212.2
Magic Chef Mcbr360W 3.6 Cubic-Ft Refrigerator	refrigerators	1	211.56
Breville Cafe Roma Espresso Machine	kitchen_appliances	1	149.99
Sunbeam 1.0 Cu. Ft. Microwave	microwaves	1	99.99
Magic Chef Mcd990B Microwave Oven	microwaves	1	99.94
LG Tall Tub Built-In Dishwasher	dishwashers	0	0
LG WaveForce 5.4 Cu. Ft. Top Load HE Washer with Heater	washers_dryers	0	0
LG 5.0 Cu. Ft. Front Load Washer	washers_dryers	0	0
Frigidaire Affinity 4.2 Cu. Ft. Front-Load Washer	washers_dryers	0	0
Bosch 5.0 Cu. Ft. Self-Clean Gas Range	stoves_ranges	0	0
Samsung 24-in Stainless Steel Built-In Dishwasher Height Adjustable Top Rack Storm Wash	dishwashers	0	0
Frigidaire Gallery 24-in Stainless-steel Built-in Dishwasher	dishwashers	0	0

Example of a report on the sales of each product

3.2 Web Service

In order to communicate within the closed eco system, our ecommerce website needs to be able to respond to certain API calls.

Our API for web services that allow stores to communicate between each other is located at /api, namely having 2 files, ‘products.php’ and ‘orders.php’. The ‘products.php’ file implements a way of returning a list of products available on our store, another method that obtains the details of a specific product and a last method that allows a store to buy a product from our store. The API calls are available online.

When returning a list of products, we make sure to check that we only return

products that are currently available (in stock):

"SELECT pid FROM product WHERE quantity > 0"
Extract products, add them into a json object which will be returned

When returning the details of a specific product (constraint: this product should only be called if it was listed in the list of products of a store; so a store should only get the details of a product from another store if that product is in stock):

"SELECT * FROM product WHERE quantity > 0"
Then, extract each column, and add it to a json object which will be returned

When attempting to purchase a product (constraint: should also be checked if it was in the list of products of a store, so the product is in stock)

"SELECT * FROM product WHERE quantity > 0"
Then, extract each column, and add it to a json object which will be returned

In our database, a special userid having the email field equal to 'OTHER_STORES' is used in order to keep track of all the purchases that were made by other stores to our store. This user is not initiated until a store buys from us. When it does, it will get inserted into the database. This user cannot be replicated by a user registering with the same email as it will fail the email validity constraint. Inserting 'OTHER_STORES' when the first foreign purchase is initiate is done by:

"INSERT INTO user VALUES (null, '$other_stores_useremail', '$other_stores_useremail', null, null, null, null, null, null)"

Thus, the user with the email 'OTHER_STORES' is unique and can be used as a global user for keeping track of foreign purchases from our website. Another user that is unique and uses the same method is the 'admin' user (email = 'admin').

After 'OTHER_STORES' has been initiated or selected, make sure there are enough products in stock in our market, even though this should not be a problem; acts like a fail-safe. Then, subtract the # of products that were bought from our market, and update the value in the database.

It grabs the count of all orders in the pendingOrders and userOrders tables and adds 1 to get the order id string to return. The count query is:

"SELECT (SELECT COUNT(*) FROM pendingOrders) AS count1, (SELECT COUNT(*) FROM userOrders) AS count2 FROM dual";

After that it generates a delivery date (which is the current date + 1 day) and then stores the transaction in the userOrder table. It also store which product was purchased in the productOrders table. It then builds the JSON obejct containing the orderID and deliveryDate and returns it to the caller.

3.3 Features

3.31 Authentication

Only an email and a password is needed in order to register, as we realize that registration forms are quite inconvenient for a majority of users.

Ajax was used in order to register or login a user, as it provides almost instant responses. As a result of the speed of the responses, we only do server-side validation for logging in or registering into the website. *authenticationService.php* is responsible for this validation process, as well as other processes like registering users, checking the logged in user, and restoring cart totals.

authenticationService API

```
POST /functionsPHP/authenticationService
params: type {login, register, logout}, email, password, password2 {for registering, password checking}
return = {
  type: "success"                    // or “error”, “”
  value: “andy@yahoo.com”        // or “logged out”
  cartTotal:”504.24”
}
```

When a user logs in, their email is stored in sessions, specifically the $_SESSION[‘email’] variable. We decided to use sessions as they provide a convenient and accessible way to store data.

3.31 Search

The search is using AJAX in order to query and produce results from the database. While this may not be very efficient when there are a lot of products to search through, this functionality offers fast query results and unique features such as letter by letter updates of your search query.

The search tool implemented also offers various graphic filters, which users will find more convenient rather than text input filters as it allows them to browse through a large variety of products in a short amount of time. The filters, as well as sorting, changing the query by a letter, or searching by a different type are all updated as fast

as the database processes the given query, without having to refresh the page.

The currently filters implemented in the search module are:

- Sorting randomly or by the lowest or highest price
- Searching by 3 different types (by name, category or product id)
- Using a range of prices (min - max)
- Using a range of product weights (min max)
- Searching by the minimum amount of products in-stock (at this store; this does not account for the availability of this product at other stores)
- Filtering the search by certain categories

This is done by calling the function 'filterSearch()' function, found in 'functionsJS/generalFuncs.js', which fires an AJAX request to 'functionsPHP/searchService.php', which in turn connects to the database and processes the request.

3.32 Recommendations by Search

The recommendation system that we built based on search is based on 2 condition. If a search query returns less than 5 results, the website allows you to relax the filters, which will extend the ranges of some filters. This might or might not produce new products.

If a search query returns more than 10 results, then the website allows you to tighten the filters, which will narrow down the ranges of some filters. This might or might not narrow down the results to less than the original results.

When relaxing and narrowing the results, the recommender system performs a formula applied to the current filters (for relaxing the query, the 'relaxRange($depthLevel, $rangeType)' function is called; for narrowing the query, the 'narrowRange($depthLevel, $rangeType)' function is called; they are both in 'functionsPHP/searchService.php').

3.32.1 Relaxing the Query

The 2 variables, 'depthLevel' and 'rangeType' are the parameters that decide how far (max) the filters will expand to and to which filters they are going to expand to in order to obtain results. With a 'depthLevel' of 3, the filters will be recalculated at most

'depthLevel' -1 times or until the number of products found using the relaxed filters will be higher than the number of products found using the original, non-relaxed filters.

The 'rangeType' determines which filters are going to be ranging:

If 'rangeType' = 1, only the range of the price filters will be recalculated.

If ‘rangeType’ = 2, the range of the price, as well as the range of the weight filters will be relaxed

If ‘rangeType’ = 3, the range of the price, the range of the weight and the minimum quantity of products in stock will be relaxed.

High level pseudo code of relaxing a query

```
function relaxRange()

filteredPriceHigh = upper limit of price range; filteredPriceLow = lower limit of price range;
While (# of ‘originalResults’ >= # of ‘filteredResults’) {
    increase current depth level
    if (depthLevel = current depth) break and return the ‘filteredResults’
    if (rangeType == 1)
        filteredPriceHigh = filteredPriceHigh * 1.1          // exponential increase
        filteredPriceLow = filteredPriceLow * 0.9
    else if (rangeType == 2)
        filteredPriceHigh = filteredPriceHigh * 1.1          // exponential increase
        filteredPriceLow = filteredPriceLow * 0.9

        filteredWeightHigh = filteredWeightHigh + ((20* current depth level)/100) * upper limit of weight range
                                                                                    // percentage increase
        filteredWeightLow = filteredWeightLow + ((20* current depth level)/100) * lower limit of weight range
    else if (rangeType == 3)
        (filtered price and weight like above AND)
         filteredMinimumQuantityInStock =  floor (original min quantity in stock - 20%  of original min
        quantity in stock)
    }
    ‘filteredResults’ = query with the newly calculated filters
    # of ‘filteredResults’ =  # of results of ‘filteredResults’
}
```

While calculating the new filters, the price range is exponentially increased, as we thought this appropriately explores a bigger area as it is going deeper than a percentage increase, which is important as we want to offer the customer more results to choose from , whether the products are cheaper or more expensive.

The weight range is adjusted by percentage as it cannot vary as much as the price from product to product of the same category.

The same applies to the minimum quantity of a product in stock, as it is adjusted by percentage.

3.32.2 Narrowing the query

When narrowing the query, the concept is opposite of relaxing the query, but it is going through similar steps. The while condition is however, quite different:

```
function narrowRange()
...
 While (# of ‘filteredResults’ < ( 65% of the # of ‘originalResults’)), apply filters, else break and return
the ‘filteredResults’ calculated in the last iteration.
```

...

When filtering, whether narrowing or relaxing the query, if the filtered results break the while condition, those filtered results will be returned, otherwise it will run all the levels and return the results of the last level. Here is the order of the relax filters called:

```
if (relaxRange(3, 1)){
  return;                                          products as a recommendation
} else if (relaxRange(3, 2)){
return;
} else if (relaxRange(3, 3)){
return;
} else if (relaxRange(5, 1)){
return;
} else if (relaxRange(5, 2)){
return;
} else if (relaxRange(5, 3)){
return;
}
```

The API of the AJAX call ‘filterSearch’, which is used for searching and producing recommendations:

```
POST /functionsPHP/searchService
params:
priceLowArg: priceLowRange, priceLowArg: priceHighRange, minQuantity: minQuantityInStock,
weightLowArg: weightLowestRange, weightHighArg:weightHighestRange, searchType: {by name, category,
pid}, searchQuery: {user entered text}, categories: {categories selected/checkboxes}, sortBy: {lowestPrice,
highestPrice}
return = {
 originalResults: original_query_results,
 originalResultCount: original_query_results_count,
 modifiedResults: original_query_results,
 modifiedResultCount: original_query_results_count,
 origFilters: originalFiltersPrettyText,
 modFilters: modifiedFiltersPrettyText,
}
```

3.32 Shopping Cart

The shopping cart is implemented in a way such that it is persistent across all pages during a session, and is also saved for logged in users, so that they have access to their shopping cart whenever they log into the website. This module also uses AJAX in order to modify the cart, so that the user get instant feedback about the status of their cart without even reloading the page.

When the session is first started, there is no cart saved to the session. As soon as a user adds an item to a cart, a session variable is created. If a user logs in, their

shopping cart gets loaded from the database (if it exists), and if a “guest cart” exists, the two get merged so that the user gets a seamless shopping experience.

Since the cart does not need to be queried for any statistics, it is saved in the session and in the database in the following JSON format, for easy access:


```
{
   "products": [
      {
         "id": "c000021",
         "name": "Samsung 24-in Stainless Steel Built-In Dishwasher Height Adjustable Top Rack Storm
Wash",
         "price": 1661,
         "img": "c000021.jpg",
         "quantity": 1
      },
      {
         "id": "c000023",
         "name": "Lamber Commercial Dishwasher With Temperature Booster",
         "price": 4534.2,
         "img": "c000023.jpg",
         "quantity": "2"
      },
      {
         "id": "c000013",
         "name": "Panasonic 1.6 Cu. Ft. Microwave",
         "price": 279.99,
         "img": "c000013.jpg",
         "quantity": 1
      },
      {
         "id": "c000014",
         "name": "Sunbeam 1.0 Cu. Ft. Microwave",
         "price": 99.99,
         "img": "c000014.jpg",
         "quantity": 1
      }
   ],
   "total": 11109.38
}
```


The total is kept tallied in order to easily access it and display it in the link in top right. When a user submits an item to the cart, Javascript performs an AJAX call to the cartService API:

cartService API

```
POST /functionsPHP/cartService
params: type {update, add, remove, checkCart}, id, quantity, data
return = {
          cart JSON object
```

```
}
```

The first three “types” perform either an update, addition, or deletion to the cart. If the user is logged in, these changes are also saved to the cart in the database. The “checkCart” type returns true if a cart exists in the session.

Once the AJAX call to this service returns, the JavaScript modifies the cart total in the upper right, and if the user is on the cart page, it updates the prices on the cart. If the user removes an item, it animates the removal of an item in a visually appealing way.

3.32 Rating System

A user that has purchased a specific items is given the option to submit a rating of that item from the product page. If the user is logged in and has a recorded order of the product, JavaScript code is echoed by PHP that allows the user to submit a rating by selecting the amount of “stars” to give the product. This code performs an AJAX request to the ratingService API:

ratingService API

```
POST /functionsPHP/ratingService
params: rating {0-5}, pid {id of the product}
return = {
        -integer value of the product rating after averaging it with the user’s submitted rating
}
```

Two columns in the product table in the database keep track of the sum of all ratings, and the count of all ratings, so that the ratings can be easily modified and calculated from the two values.

Each user’s specific rating of an item is also stored in a table so that if a user chooses to update their rating, the service subtract their previous rating from the product’s sum of ratings, and adds on the new rating. It then proceeds to calculate the current rating (outside of the database) to return to the AJAX so that it can display the new rating on the page without refreshing.

3.33 User Profiles

Registered users must entrust some personal information in order to be allowed to make purchases on the Black Market. This information can be viewed and edited at

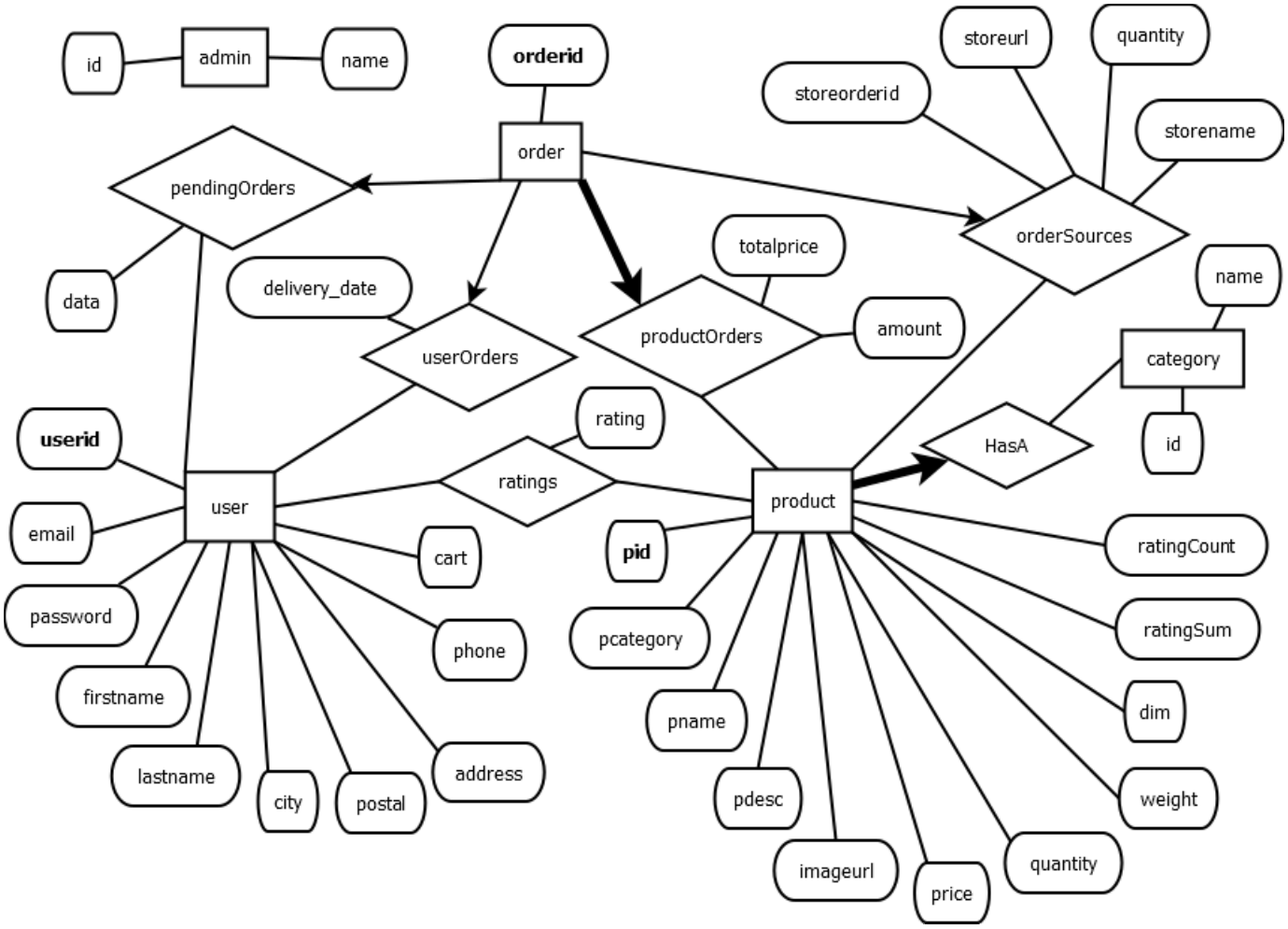
any point by the user by updating their profile, found by clicking their email in the top right hand corner of the page. Updated profiles are saved using an AJAX post to the internal service *userProfileService.php* which updates the row in the database using the passed values. In addition to personal information, users are also able to see a listing of their past transactions. This data is displayed in a table with the same format as the tables found in the management module. Delivery dates for products purchased from other stores maintain accuracy with a GET request to the store in question:

```
GET /orders/:id
return = {
  "delivery_date": "2013-01-01"
}
```

Appendix A - Citation

- JQueryUI
  http://jqueryui.com/
- JQuery Tablesorter 2.0
  http://tablesorter.com/docs/
- FirePHP firefox extension
  http://www.firephp.org/

Appendix B - ER diagram


id
admin
name
orderid
storeurl
quantity
storeorderid
storename
order
pendingOrders
orderSources
totalprice
name
data
delivery_date
productOrders
amount
userOrders
category
rating
HasA
id
userid
ratings
user
product
email
cart
pid
ratingCount
password
phone
pcategory
ratingSum
firstname
pname
dim
lastname
city
postal
address
pdesc
weight
imageurl
price
quantity