COMP0178: Database Fundamentals

Group 4
Zuzanna Sosnowska
Kyriaki Charalambous
Omiros Trypatsas
Maria Usova-Sinclair

The auction system

The brief asks us to design a consumer-to-consumer online auction system; an online application that permits registered users to either sell or purchase items. Upon registration, a new user is asked to choose either a 'seller' or 'buyer' account type that, in turn, determine their capabilities on the website. A 'seller' account type is able to create auctions to sell their items, with their chosen start price, reserve price and end date. In addition, they are able to watch others users bid on their auctions and review their active and past listings. A 'buyer' type account can browse active auctions and place bids at them, and, in addition, they can add auctions to their watchlist, in order to receive relevant notifications regarding that item.

In order to achieve such capabilities, a website front is not sufficient. We require a functioning database system that will capture all relevant data about the users and the auctions. The database design process is summarised within this report. We begin with a conceptual data model in the form of an Entity-Relationship diagram that captures all the entity types (objects of interest), their attributes (characteristics and properties) and the relationships between them. Using this ER diagram, we are then able to translate this schema into a logical data model. For a relational database system, this consists of formulating the appropriate tables by studying the relationships and their multiplicities determined by the ER diagram.

ER model

The Entity-Relation for the auction system is depicted in Figure 1.

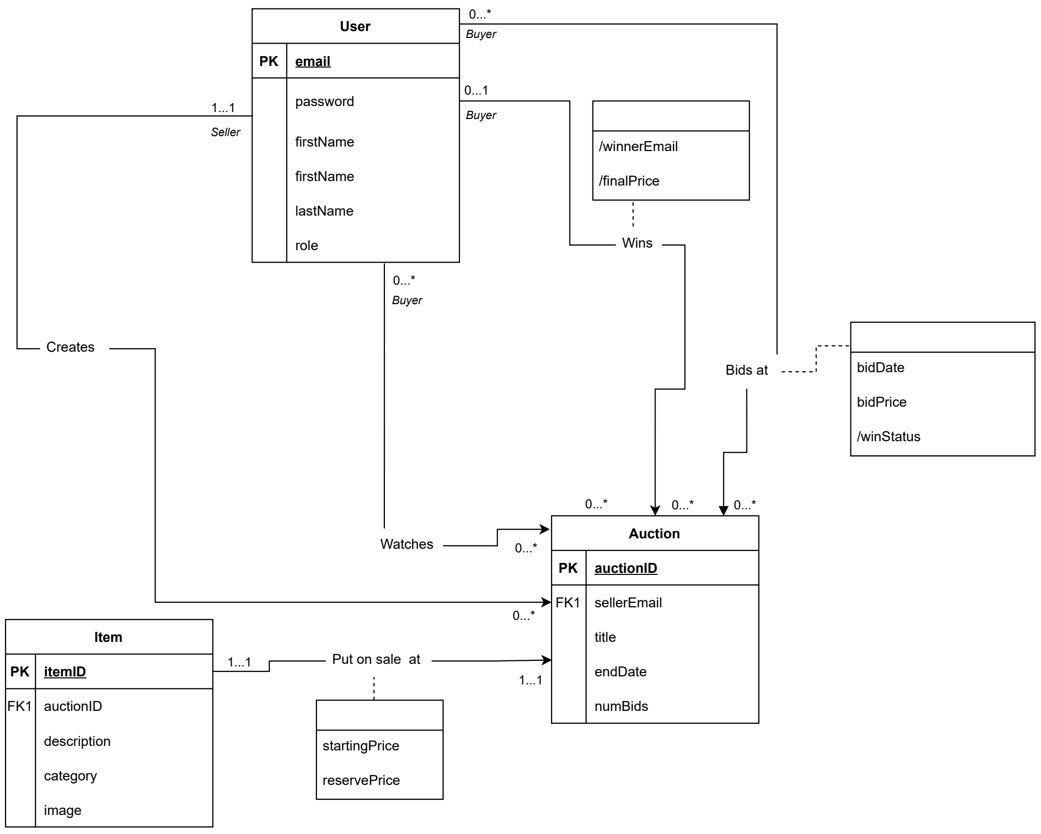


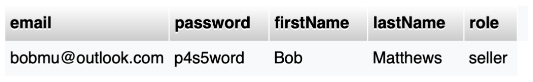
Figure 1: ER diagram

1 ER translation for relational data models

We individually consider the attributes and relationships identified in the ER model.

1.1 Strong and weak entities

Let us consider the three entities identified in the ER model, namely ‘User’, ‘Auction’ and ‘Item’. Recall that strong entities are those that do not depend on the existence of other entities for existence. In the case of the auction system, the only strong entity is ‘User’. Its associated table is depicted in Figure 2. With the assumption that a user may register only one account per email address, we can assign ‘email’ as the primary key for the table.



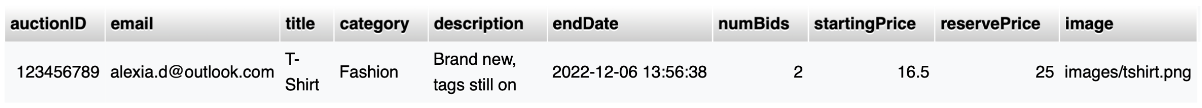
email	password	firstName	lastName	role
bobmu@outlook.com	p4s5word	Bob	Matthews	seller

Figure 2: User table: ‘email’ PK

Both ‘Item’ and ‘Auction’ are weak entities. In particular, the existence of ‘Auction’ relies on the existence of a seller. In turn, the ‘Item’ entity relies on the existence of ‘Auction’. One option is to create a separate table dedicated to each entity. In this case, we ought to include (seller) ‘email’ as a foreign key in the ’Auction’ table and ‘auctionID’ as a foreign key in the ’Item’ table. However, upon studying the 1:1 relationship between the ‘Item’ and ‘Auction’ entities, we opted to combine the two entities into a single table. In this case, it is sufficient to include (seller) ‘email’ as a foreign key to link each auction to the seller that created it.

1.2 1:1 Relationships

The ER diagram identifies that ’Item’ has a one-to-one relationship with ’Auction’, since only one item may be sold per auction. Moreover, the two entities are dependent on the existence of the other for existence; that is, there is *mandatory participation* on both sides. It follows that these two entities are best combined into a single table that includes both entity attributes and relationship attributes. In this instance, we no longer require both ‘itemID’ and ‘auctionID’; a single unique identifier is sufficient. The resultant table ’Auction’ is as follows:



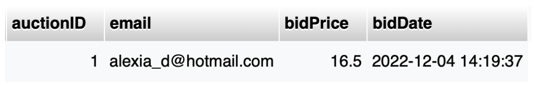
auctionID	email	title	category	description	endDate	numBids	startingPrice	reservePrice	image
123456789	alexia.d@outlook.com	T-Shirt	Fashion	Brand new, tags still on	2022-12-06 13:56:38	2	16.5	25	images/tshirt.png

Figure 3: Auction table: ‘auctionID’ PK, ‘email’ FK

It is important to note the following. When it comes to storing information about the bids, we have two possible options. Firstly, we could store each individual ’Bid’ placed by a particular user as a separate row in the database. In this case, every bid requires a unique ‘bidID’ to identify it, as the same user can place multiple bids on the same item. The disadvantage of this approach is a very large amount is additional data will be stored in the database, which would use a lot of memory. We opted for a different approach, where instead we generate a single row per (user, item) tuple and simply update the bid price every time that particular user places a higher bid on that particular item. This will save a lot of memory whilst still storing all the relevant information required to make the bidding process work. However, since we are not storing individual bids in the ’Bid’ table, but only the (user, item) tuples, we do not have a way to calculate the number of times an item has been bid on. Therefore, we require a ‘numBids’ column in the ’Auction’ table, whose value is increased by 1 every time a new bid is placed on the item. This process is further explained in Section 3.4.

1.3 *:* Relationships

There are two *:* relationships to be considered: ‘watches’ and ‘bids at’. In general, a *:* relationship requires us to create a dedicated table that includes the appropriate foreign keys and the relationship’s attributes. The resulting ’Bid’ table is depicted in Figure 4.

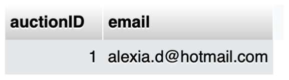


auctionID	email	bidPrice	bidDate
1	alexia_d@hotmail.com	16.5	2022-12-04 14:19:37

Figure 4: Bid table: (‘auctionID’, ‘email’) PK

To translate this into our relational database, we include the ‘auctionID’ attribute from the ’Auction’ table and the (bidder) ‘email’ attribute from the ’User’ table as foreign keys. Given that we require both the auction and the user who places the bid in order to determine the relevant bid price and date, the table must have a composite primary key.

Analogously, we created a ‘Watchlist’ table which includes ‘auctionID’ and ‘email’ columns as foreign keys. An entry is added to the table if a user chooses to add a particular auction to their watchlist, and it is deleted if the auction is removed from the user’s watchlist. As this relationship has no additional relationship attributes, these are the only two columns we require. The final table is as follows.



auctionID	email
1	alexia.d@hotmail.com

Figure 5: Watchlist table: (‘auctionID’, ‘email’) PK

1.3.1 1:* Relationships

We have to consider two distinct 1:* relationships present in the ER model: ‘creates’ and ‘wins’. In the relational database, a 1:* relationship may be modelled analogously to a parent-child relationship.

In the auction system, one seller may create 0 to infinite auctions. On the other hand every auction requires at least one and maximum one seller. It is important to note that the existence of the ‘Auction’ entity is reliant upon the existence of ‘User’ (seller); thus we have *mandatory participation on one side.* In particular, the ’Auction’ table requires the ‘email’ attribute from the User table included as a foreign key under the column name ‘sellerEmail’, as shown in Figure 3. This way, all personal information about the seller of a particular item may be found by joining the two tables on User.email = Auction.sellerEmail.

In addition, the ER table identifies a 1:* relationship, namely ‘wins’, between ‘User’ (buyer) and ‘Auction’ for which participation is *optional* on both sides. If the auction receives no bids or the final price does not reach the reserve price set by the seller at creation, the auction will have 0 winners. Otherwise, the auction will have at most one winner – the buyer with the highest bid. Note that we do not need to store the associated relationship attributes ‘winnerEmail’ and ‘finalPrice’ in the ’Auction’ table, as these values may be calculated by executing appropriate SQL queries on the Bid table (e.g. “SELECT MAX(bidPrice) FROM Bid WHERE auctionID=1;” to find the highest bid).

2 Normalisation

Normalisation can be explained as the process of eliminating unnecessary duplicated data that may cause update, insertion and deletion anomalies. For our purpose, we require all tables to be in third normal form (3NF).

By definition, a table is in 1NF if no cell in the table contains repeated groups, namely multi-valued attributes or a set of attributes. Moreover, each column’s name must be unique. A table is in 2NF if the table is already in 1NF, and, in addition, there are no partial dependencies on the primary key in the table.

More precisely, this means that there are no non-primary key attributes that are fully determined by a part of the primary key. Finally, a table is in 3NF if it is already in 2NF, and there are no transitive dependencies.

The first table we consider is

Auction (auctionID PK, email FK, title, category, description endDate, numBids, startingPrice, reservePrice, buyNowPrice, image)

where 'auctionID' is the primary key. To show that this table is in the third normal form, we start by defining its functional dependencies, as these show the relationship between attributes in the table.

auctionID → email, title, category, description, endDate, numBids, startingPrice, reservePrice, buyNowPrice, image

It follows that the requirements for the first fundamental form are met as, firstly, every column name is unique, and, in addition, there are no multi-valued attributes. Moreover, it does not have a composite primary key; thus, it cannot have partial key dependencies. As the tabel is in 1NF and has no partial key dependencies, we conclude that the table is also in 2NF. Lastly, since it is in 2NF, and there are no indirect relationships between attributes that may cause a functional dependency, this table is also in 3NF.

Our second table is

Bid (auctionID FK, email FK, bidPrice, bidDate)

with 'auctionID' and 'email' as the composite primary key. Its functinoal dependency is

auctionID, email → bidPrice, bidDate

Evidently, the table meets the conditions for the first normal form; each column name is unique and there are no multi-valued attributes. The table is also in 2NF, since neither 'bidPrice' nor 'bidDate' can be deduced from 'auctionID' or 'email' alone. Lastly, the table is also in 3NF as we cannot deduce 'bidPrice' from 'bidDate' and vice versa; thus, there are no transitive dependencies between attributes.

The third table we consider is

User (email PK, password, firstName, lastName, role)

Its functional dependencies are as follows:

email → password, firstName, lastName, role

All entries are single-valued and all column names are unique; therefore, the table is in 1NF. In addition, as the table has a singular primary key, it is automatically in 2NF. Finally, we cannot deduce any attributes related the user from other attributes. In other words, there are no functional dependencies by the virtue of transitivity, and so, the table is indeed in 3NF.

Our final table is

Watchlist (auctionID FK, email FK)

This table is an example of a junction table; its key purpose is to represent the *:* relationship between the 'Auction' and 'User' tables. Note that the table is only composed of two columns, namely 'auctionID' and 'email', where both columns are foreign keys. It follows that we do not have any non-key attributes within a table; therefore, no functional dependencies can exist. Nevertheless, this table is essential for meeting the requirements of normalisation for our particular database.

3 SQL queries

3.1 User account

3.1.1 Register

To use the auction system a user must first sign up using a registration form. The input data values for role, email, password, first name and last name first undergo dynamic JavaScript-based validation of the data format. The form alerts users of invalid data before it is submitted. Once the form is ready to submit to the database, this query:

```
SELECT * FROM User WHERE email= '$email' LIMIT 1;
```

where the PHP email variable is an input value, is used to determine whether this user has already registered. In case they have, they receive an informative message. If they have not, this query:

```
INSERT INTO User(email,password,firstName,lastName,role)
VALUES ('$email','$password','$firstName','$lastName','$accountType');
```

which includes PHP variables connected to the HTML input tags, is used to insert their information in the 'User' table of the auction system. If this query is executed successfully they will be registered. Otherwise, the user is informed that registration has failed.

3.1.2 Login

To login into their account users must complete a form with their email and password. Similar to registration, the input values are validated using JavaScript and once the data is ready for submission, this query:

```
SELECT * FROM User WHERE email= '$email' LIMIT 1;
```

is used to identify a user account stores in the 'User' table. If one is present, the user will be logged in; otherwise, they will be informed that registration is required.

3.1.3 Change account information

Both account types, namely 'seller' and 'buyer', are able to amend their personal information. In particular, the user can change their first name, last name or password, or a combination of these. In order to change their password, the user is asked to enter a new password and, in addition, a password repeat. If the two inputs match, the request can be carried out. The query used to update a new first name of an existing user in the table is

```
UPDATE User SET firstName = '$newFirstName' WHERE email = '$email';
```

Analogously, the query used to update a new last name of an existing user in the table is

```
UPDATE User SET lastName = '$newLastName' WHERE email = '$email';
```

Lastly, provided that the user has entered a valid password, and that the password repeat is a match, the query used to update the password is

```
UPDATE User SET password = $newPassword WHERE email = '$email';
```

3.1.4 Notifications

After an auction is finished, the user with the maximum bid and the seller who created the auction need to be notified of the auction outcome. The item is succesfully sold if (a) the auction has received at least 1 bid and (b) the maximum bid is greater or equal to the reserve price, if one has been set by the seller at creation. Firstly, we need to check whether the user has a 'buyer' or 'seller' account type, in order to present the appropriate notification. The query that checks the role of the user is

```
SELECT role FROM User WHERE email = '$email';
```

Upon identifying that the role of the user is a buyer, we require a table view that allows us to see all the usual item related data from the 'Auction' table and, in addition, the max bid placed on every item and the buyer who has placed said bid. The latter two values can be deduced from the 'Bid' table. The query used to achieve this is

```
WITH MaxPrice AS (
    SELECT auctionID, max(bidPrice) AS maxPrice FROM `Bid` GROUP BY auctionID)

SELECT *
FROM Auction
INNER JOIN (
    SELECT MaxPrice.auctionID, Bid.email AS bidder, MaxPrice.maxPrice, Bid.bidDate
    FROM Bid, MaxPrice
    WHERE Bid.auctionID = MaxPrice.auctionID
    AND Bid.bidPrice = MaxPrice.maxPrice) AS MaxTable
    ON MaxTable.auctionID = Auction.auctionID
```

On the other hand, if the role of the user is that of a seller, we want to view a table that show the maximum bid for every auction that (a) has ended and (b) was created by that particular seller. This can be checked by comparing the email stored in the 'Auction' table with the email that the user used to log into the website and by comparing the end date and time of the auction to the current date and time. Additionally, we want this table to include the auctionID of the relevant items as well as the usual item information found in the 'Auction' table (title, description etc.) Finally, we order the table by the end date of the auction. The query used to achieve this is

```
SELECT MAX(Bid.bidPrice) AS 'max', Bid.email, Bid.auctionID, Auction.auctionID,
Auction.title, Auction.description, Auction.reservePrice, Auction.endDate,
Auction.image
FROM Bid
INNER JOIN Auction
ON Auction.auctionID=Bid.auctionID
WHERE Auction.email = '$email' AND Auction.endDate < now()
GROUP BY Auction.auctionID
ORDER BY Auction.endDate
```

Then, using PHP code, the appropriate message is presented to the user based on their role.

3.2 Create auction

A vital part of creating a new auction is generating a new auction ID, used as the primary key in the 'Auction' table and a foreign key in the 'Bid' table, in order to uniquely identify that particular auction. Our website generates the auction ID by incrementing the most recently generated auction ID by 1. If, however, there is no auction stored in the table, that is, this the first auction ever created, then the auction ID is set to be 1. The query used to check if there exists an entry in the table is

```
SELECT *
FROM Auction
ORDER BY auctionID DESC
LIMIT 1;
```

which allows us to obtain the row of the item which holds the maximum auction ID. Then, using PHP code, we can isolate the auction ID and either increment it by 1, if the string is not empty, or set it equal to 1, if the string is empty.

After generating the auction ID, we need to store the data entered by the user, that correspond to the item, inside the 'Auction' table. The mandatory fields of the 'Create auction' form are 'Title of auction', ‘Category’, ‘Starting price’, ‘Image Path’ and ‘End date’; ‘Detail’ and ‘Reserve Price’ are optional. In particular, using PHP code, we are able to check if the reserve price string is empty or not. In the first case, we set reservePrice column in the Auction table equal to zero using the query

```
INSERT INTO Auction(auctionId,email,title,description,
    category,startingPrice, reservePrice,endDate,image,numBids)
VALUES('$auctionID','$email', '$auctionTitle','$auctionDetails','$category',
    '$startPrice', '0','$end_Date','".$targetFilePath."', '0');
```

In the second case, where the seller opted to set a reserve price, the query is

```
INSERT INTO Auction(auctionId,email,title,description,category,
    startingPrice,reservePrice,endDate,image,numBids)
VALUES('$auctionID','$email', '$auctionTitle','$auctionDetails',
    '$category','$startPrice', '$reservePrice','$end_Date','".$targetFilePath."', '0');
```

3.3 Browse

Both registered and non-registered users of the website are able to view the Browse page. At first, when no search, filter or sort option are applied, the user is able to view the whole list of items that are or were available. As a default, products are ordered from the longest expiry date at the top of the page to those that have ended at the end of the list .

To make all functionalities work we have to consider multiple different cases. Firstly, in the case that the user wants to search for an item within a particular category and, in addition, possibly using a keyword. To show the appropriate list of products, we first need to check the price that should be displayed on the browse page, which depends on the number of bids placed on the item. More precisely, if the number of bids per particular item is 0, the current price should be set to its starting price. If, instead, the number of bids is greater than 0, then the current price is equal to the value of the highest bid, found the ’Bid’ table, which in our case is called ‘maxBid’.

We first select the maximum bid price for each item, in the column titled ‘maxBid’, from the ’Bid’ table, using the select query

```
SELECT MAX(bidPrice) AS maxBid, auctionID as a
FROM Bid
GROUP BY auctionID) AS b
```

Next, we join the resulting table to the ’Auction’ table, which has the remaining relevant information about each item, as follows:

```
SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice FROM Auction LEFT JOIN (
    SELECT MAX(bidPrice) AS maxBid, auctionID as a
    FROM Bid
    GROUP BY auctionID) AS b
ON b.a=Auction.auctionID
WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
AND category = '$category'
ORDER BY endDate DESC
```

The latter part of the query allows us to select and display correct items that match the category requirement and, in the case that the search bar was also used, match the requirement of having the keyword entered by the user in either the title or the description of the auction. Additionally, if the user also wants to implement the sorting function, then the ORDER BY function should be added to the SQL query employed. To sort items by ‘Price low to high’ we need to set the ORDER BY to be by the ‘currentPrice’ column in ascending order, so that items with the lowest price are shown first.

```
SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
FROM Auction
```

```
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
        ON b.a=Auction.auctionID
        WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    AND category = '$category'
    ORDER BY currentPrice ASC
```

To sort by 'Price high to low', the ORDER BY is set to be by 'currentPrice' in descending order, which starts from the items with the highest price.

```
    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    AND category = '$category'
    ORDER BY currentPrice DESC
```

To sort by 'Soonest Expiry', the ORDER BY is set to be by the column 'endDate' in an ascending order. Here, the auctions that have already ended or are due to end soon are shown first.

```
    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    AND category = '$category'
    ORDER BY endDate ASC
```

Lastly, if the user does not want to filter their search by item category, the query used is

```
    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID  as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%'
    ORDER BY endDate DESC
```

The structure of this query is analogous to previous queries, with the only difference being that the WHERE clause does not impose conditions on the column 'category'.

The same goes for the queries listed below that consider various keyword and ordering combinations, but do not consider any particular category of items.

```
    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    ORDER BY currentPrice ASC

    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    ORDER BY currentPrice DESC

    SELECT *, IF (numBids=0, startingPrice, maxBid) AS currentPrice
    FROM Auction
    LEFT JOIN (
        SELECT MAX(bidPrice) AS maxBid, auctionID as a
        FROM Bid
        GROUP BY auctionID) AS b
    ON b.a=Auction.auctionID
    WHERE (Auction.title LIKE '%$keyword%' OR Auction.description LIKE '%$keyword%')
    ORDER BY endDate ASC
```

3.3.1 Pagination

To create a working pagination, we first need to define the number of products we wish to show on the Browse page. To do so, we construct a variable called `initial_page` that is responsible for storing the initial number of pages. This translates into the relevant SQL query through the SQL LIMIT function. In particular, it will limit the number of products selected from the 'Auction' table to the value required.

```
$total_pages_sql = "SELECT COUNT (*) FROM Auction";
```

Secondly, the above SQL statement is initiated to find the total number of rows in the 'Auction' table, which corresponds to the total number of auctions in existence. This value is used later, in order to calculate the required number of pages.

```
$offset = $initial_page . ',' . $results_per_page;
```

The variable `offset` is created to define the LIMIT in the final SQL statement. If we are considering the first page, the variable

```
$initial_page=0
```

as we want to select products from the top of the 'Auction' table and, since

```
$results_per_page=10
```

to the 10th row of the 'Auction' table.

```
$sql = $sql.'LIMIT '.$offset;
```

Lastly, we set the `sql` variable to be a combination of `sql` that is a selection of particular products, say, only products in the Fashion category (see Section 3.3), with the number of rows (auctions) that match the requirements. These product will now be displayed on the Browse page.

3.4 Place bids

To place a bid, the user enters their chosen bid value as input and commits by pressing the 'Place bid' button. However, before their bid is recorded in the 'Bid' table, we must check it is higher than the current price of the product.

The current price of the product is a derived data element; that is, we are able to calculate it from other data stored in the database. In the case when no bids have been placed on the item, the current price is simply the starting price set by the seller at creation, which can be found under 'startingPrice' in the 'Auction' table. Otherwise, it is equal to the highest bid placed, which in turn can be calculated from the 'bidPrice' entries in the 'Bid' table.

It follows that we must first check the number of bids placed on the item using the query

```
SELECT numBids FROM Auction WHERE auctionID = $item_id;
```

which returns an integer value. If the query returns 0, we use the select query

```
SELECT startingPrice FROM Auction WHERE auctionID = $item_id;
```

to obtain the starting price for the product, which we in turn assign to the variable `current_price`. On the other hand, if `numBids>0`, we use

```
SELECT MAX(bidPrice) AS max FROM Bid WHERE auctionID = $item_id;
```

This selects the highest bid placed on that product from the 'Bid' table, and we can in turn assign the value of `max` to our variable `current_price`.

On the assumption that the bid value is indeed higher than the current price, we need to record the bid in the database. First, we need to increment the value of 'numBids' by one using the action query

```
UPDATE Auction SET numBids = numBids + 1 WHERE auctionID = $item_id;
```

Next, we check whether the user has previously placed a bid on that particular item using the select query

```
SELECT * FROM Bid WHERE auctionID = $item_id AND email = '$email';
```

We use the `mysqli_num_rows()` function in PHP in order to deduce if that particular `(email, auctionID)` tuple exists in the 'Bid' table. If such row exists, we update the 'bidPrice' and 'bidDate' columns with the user's latest bid value and the current timestamp as follows:

```
UPDATE Bid SET bidPrice = $bid, bidDate = now()
WHERE auctionID = $item_id AND email = '$email';
```

In the case when `mysqli_num_rows()` returns 0, we must create a new row as

```
INSERT INTO Bid VALUES ($item_id, '$email', $bid, now());
```

Lastly, in order to notify the user than this item has been added to their 'My Bids' page, we use the select query

```
SELECT title FROM Auction WHERE auctionID = $item_id;
```

to obtain the title of the auction, which we insert into the "Bid successful! `title` has been added to My Bids."

3.5 Listings

The 'My Listings' and 'Past Listings' pages are only accessible for users with a 'Seller' account type. The 'My Listings' page displays the auctions created by the current user that are still active; the 'Past Listings' page displays those auctions that have now ended.

For the 'My Listings' page, the select query

```
SELECT *
FROM Auction
WHERE email = '$email$' AND endDate>=now()
ORDER BY endDate;
```

selects the auctions created by the current user that are still active and, in addition, orders the results such that the auctions the earliest expiry appear first.

Analogously, to fill the 'Past Listings' page, the query

```
SELECT *
FROM  Auction
WHERE email = '$email' AND endDate<now()
ORDRE BY endDate DESC;
```

selects the auctions created by the current user that have now ended and orders them such that the most recently ended auctions appear at the top of the page.

3.6 My Bids

The My Bids section of the website is only accessible to users with a 'Buyer' account type. It lists those auctions that the current user has placed a bid at and are still active. This allows the user to keep track of the auctions that they are taking part in. The format of the listing is largely similar to the Browse page; however, instead of solely stating the current price of the item, the listing display the current price, namely 'Highest Bid', and the current user's latest bid, 'Your Bid'. Auction where the current user has been outbid are displayed in red. To obtain the correct auctions, we employ the select query

```
SELECT *
FROM Auction
INNER JOIN Bid
ON Bid.auctionID = Auction.auctionID
WHERE Bid.email = '$email' AND endDate>now()
ORDER BY Bid.bidDate DESC;
```

for which the `WHERE` clause isolates the current user's auctions that are still active, and the `ORDER BY` clause ensures that the auctions that the user has bid at most recently appear at the top of the page.

3.7 Watchlist and email notifications

For all active auctions, buyers have the option to add and remove themselves from a watchlist for that particular auction. This function requries the following queries:

```
INSERT INTO Watchlist (auctionID, email) VALUES ('$item_id', '$email');

DELETE FROM Watchlist WHERE email='$email' and auctionID='$item_id';
```

in order to add and remove entries, respectively. Specifically, the auctionID and the email of the buyer are added and removed from the 'Watchlist' junction table. When buyers are added to the 'Watchlist' table they start receiving email notifications when buyers bid or raise their bids. Every time a bid is added, this query:

```
SELECT * FROM Watchlist where auctionID = '$item_id';
```

is used to determine whether someone is watching this specific auction. If a user is subscribed to the watchlist, then three types of emails can be sent, depending on the user watching. If the person who is placing the bid is also watching i.e. the current session email is found in the 'Watchlist' table, they will be informed that they currently have the highest bid. To determine if other people who bid are watching, this query:

```
SELECT * FROM Bid WHERE auctionID = $item_id AND email = '$email1';
```

is used, where results are returned if the email column entry of the 'Watchlist' table matches an email found in the 'Bid' table for a specific auction. If the above query finds a match successfully, then bidders will be informed that they have been outbid. Otherwise, if buyers are watching but have not bid, they will be informed that bids were placed on the item they are watching.

3.8 Recommendations

There are two dominant types of recommendation systems: content-based filtering and collaborative filtering. Content based filtering uses the item's attributes and characteristics to determine whether that particular item ought to be recommended to the user. On the other hand, collaborative filtering uses the users' interactions to determine what items they should be recommended. The idea is that people who purchase similar items have similar tastes; therefore, they are likely to be interested in similar items in the future.

For the auction system, we provide three different types of recommendations; two are content based, and the other employs collaborative filtering. In particular, the user is recommended:

- auctions that have the highest number of different users participate in during the past week ("trending" products)
- recommendations based on the latest items that the user bid on (using collaborative filtering)
- items from the same category as the latest items bid on by the user.

The first type are trending items. This is a content based recommendation that follows the idea that items with the highest number of bids are most "on trend", and hence more desirable. To obtain listings of the "trending" items, we use the select query

```
WITH bidcount AS (
    SELECT auctionID, COUNT(*) AS c FROM Bid
    WHERE bidDate BETWEEN date_sub(now(), INTERVAL 1 WEEK) AND now()
    GROUP BY auctionID)
SELECT * FROM Auction
INNER JOIN bidcount ON Auction.auctionID = bidcount.auctionID
WHERE endDate>now()
ORDER BY c DESC
LIMIT 3;
```

The first column 'auctionID' of the resulting table is a list of all the auctions in the 'Bid' table (auctions that have at least one bid). The other column is the count of how many times that item appears in the table, which is equivalent to how many different users placed a bid at that auction. In addition, to obtain the most currently popular items, we limit the entries to only the bids that have been placed in the past week. In the second part of the query, we join the 'bidcount' table with the 'Auction' table to obtain the additional information about the popular auction that we require to publish the listing. We limit the auctions to only those that are still active (`endDate>now()`) and order them by popularity (number of users who bid on them). In this section, we chose to display items that the user has previously bid on if they also happen to be the current trending items.

The second part of the user's recommendations is based on collaborative filtering. In order to find fitting items, we employ the following SQL query:

```
WITH bids AS (
    SELECT auctionID
    FROM Bid WHERE email = '$email$'),

    other_users AS (
        SELECT email, Bid.auctionID
        FROM Bid, bids
        WHERE Bid.auctionID = bids.auctionID),
    recs AS (SELECT Bid.auctionID, COUNT(Bid.auctionID) AS c
             FROM Bid, other_users, bids
             WHERE Bid.email = other_users.email AND Bid.auctionID NOT IN (SELECT auctionID
    FROM Bid WHERE email = '$email')
             GROUP BY Bid.auctionID
             )

SELECT *
FROM Auction, recs
WHERE Auction.auctionID = recs.auctionID
AND endDate>now()
ORDER BY c DESC
LIMIT 5;
```

The first `WITH` select statement finds the items in the 'Bid' table that the current user has previously bid on and stores their corresponding auctionID in a temporary "bids" table. We then select the email addresses of users in the 'Bid' table, who have bid on the same items as the current user; that is, the items stored in the "bids" table. Note that this will also select the email address of the current user, but this will be excluded further in the query. The users' email addresses are stored in the "otherusers" table. The third component of the query, namely "recs", selects the items in the 'Bid' table that users in "otherusers" have previously bid on and exclude those items that the current user has already bid on. In addition, we select how many times each item appears in the result. This allows us to eventually determine the top items that are most suited for the current user. The very last part of the query selects the relevant information for these items from the 'Auction' table and excludes the auctions that have already ended, in order to display the items' listings in PHP. In addition, we order the result by the count, from most to least common, and limit to the 5 most relevant results.

The final part of the recommendation system recommends items from the same category as items previously bid on by the current user. Firstly, we use the select query

```
SELECT DISTINCT category FROM Auction
INNER JOIN (
    SELECT auctionID, bidDate
    FROM Bid
    WHERE email = '$email') AS bids
ON Auction.auctionID = bids.auctionID
ORDER BY bids.bidDate DESC
LIMIT 3;
```

Here, we first select all the items that the current user has bid on and order them from most to least recently bid on. Out of these items, we select three distinct categories. We store the result in a PHP variable `category`. Using this variable, for each iteration of `category`, we employ the select query

```
SELECT DISTINCT auctionID, title, description, numBids, image, endDate, startingPrice
FROM Auction
WHERE category = '$category'
AND Auction.auctionID NOT IN (
        SELECT auctionID
        FROM Bid WHERE email = '$email')
AND endDate>now()
LIMIT 3;
```

which finds three distinct active auctions in that category that the user has not yet bid on.

4 Link to a project video

`https://youtu.be/9b9AaU9wNP0`