COMPGC06 Databases and Information Systems: Coursework

Michael Aring, Luke Harries, Kai Klasen, Christina Kronser

March 16, 2018 - Group 20

Table of Contents

---

1 Project Overview

This report describes the design and implementation of a functioning database structure for an online auction system for winter sports equipment, called Bidly. Main features include three user roles - administrator, seller and buyer - with different privileges. Whereas a seller can create an auction for an item and receive reports on its progress and viewing traffic, buyers can search for products based on its name and descriptions as well as sort the resulting list by ascending and descending bids, the auctions' end dates, seller ratings and by different categories. After spotting an interesting object, they can bid on it and simultaneously see all other bids that were placed on this item as well as a countdown until the auction ends. Buyers can furthermore place auctions on their watch list, thereby enabling email updates on new bids placed for this item and notifications in case they are outbid. An additional dashboard provides a good overview on this information, too. By bidding on items they enable a smart algorithm to make further auction recommendations based on the auctions they engaged in already. The system keeps track of the auctions, finishes them at their given ending time and informs both the seller and highest bidder. All users may view a report summarising their buying and selling activities and are invited to leave feedback after auctions which can be seen by all platform users.

The live hosted version of the system can be viewed at: http://52.232.32.194/

2 Video Link

https://docs.google.com/document/d/1ZjgUpgGzacrIOEJsoAdA1VssJeu3wJr5BpplTQjuv30/edit?usp=sharing

3 Entity Relationship Diagram

Figure 1 shows the Entity Relationship Diagram we used for the auction system. It underwent an iterative process to depict how our database objects ("entities") relate to each other. Assumptions that were made are that buyer and seller do not need separate accounts as one user could be both. Only the administrator role encompasses functionality that is restricted to normal platform users and is therefore marked through a separate table. Moreover, one user can add more than one item as auction and feedback can only be given after the auction has ended.

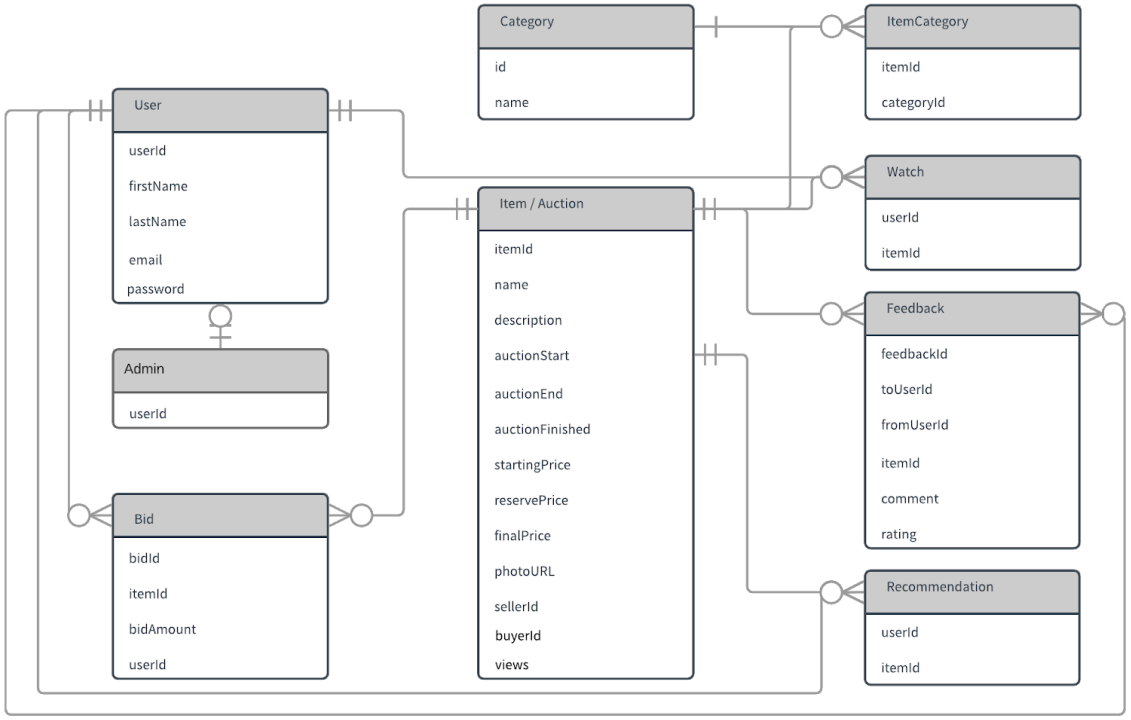

Figure 1: Entity Relationship Diagram

4 Database Schema

Figure 2 shows the database schema for the auction system, acting as a blueprint of how the relational database is constructed. It shows a number of key constraints and functions by indicating unique, non-optional and auto-incremental variables. The links between tables are displayed through primary and foreign keys. To ensure referential integrity the database features a number of foreign key constraints. The admin table is the only table with no foreign key dependency, and tables such as the ones for *itemCategory*, *watch* or

recommendation make use of a compound key, consisting of two foreign keys that uniquely identify an entity occurrence.

To provide appropriate abstraction and encapsulation we made use of a PHP HTTP REST API and SQL stored procedures. The REST API enables our application to cooperate with the database using REST concepts. It is an easy way to create, read, update or delete information by using a series of nested endpoints. Each endpoint performs a given database operation, manipulating the retrieved data if necessary, and returning it to the caller.

The SQL stored procedures allow us to store SQL queries that are used several times with the ability to pass parameters to them. Thereby, the queries act according to current needs and can handle the different requests that are made based on the input parameters. Another advantage is that query changes can be made in a single location, providing an efficient layer of abstraction and avoiding unnecessary code repetition. As every query throughout the auction system is implemented through one or more stored procedures, it enables the database engine to optimise the procedure's query plan.

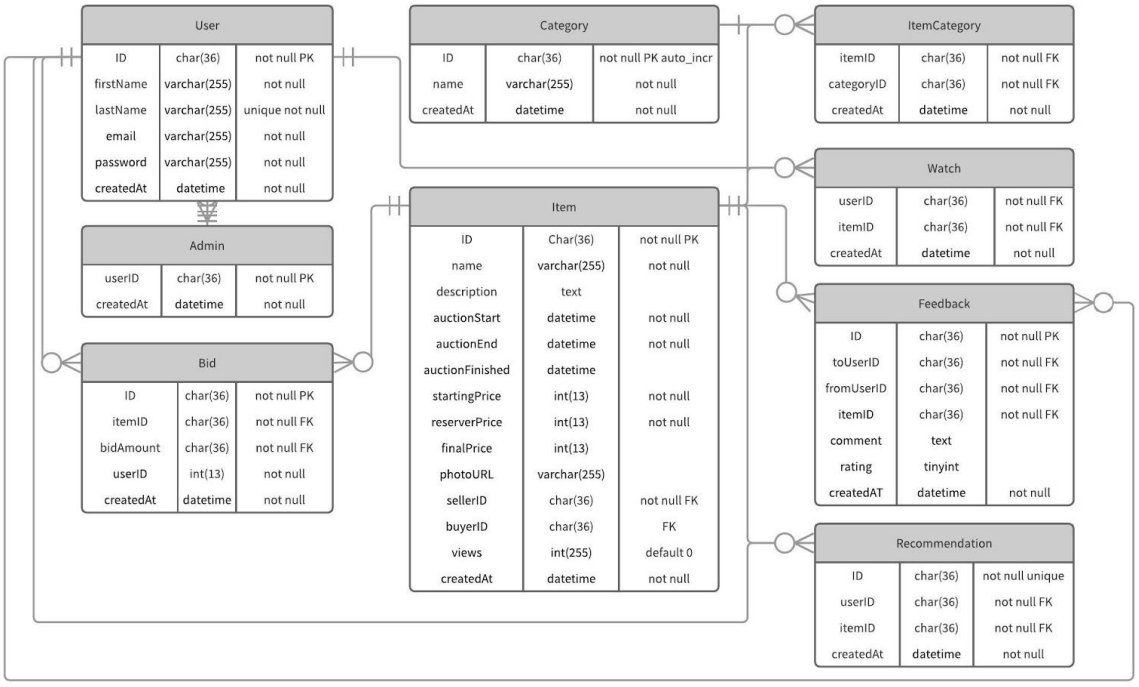

Figure 2: Database Schema

5 Application Architecture

In addition to the standard schema definitions, the system makes use of several web architecture paradigms. Firstly, we implemented a multi-level system architecture by designing it in several layers. A JavaScript frontend combined with the PHP HTTP REST API and a MySQL database with stored procedures ensure an appropriate level of abstraction and encapsulation allowing for easier design changes. We also use a model-view-controller (MVC) architecture to separate input, processing and output of the application. The database

queries, the resulting PHP functions and the models follow a Data Access Object (DAO) architecture to encapsulate and abstract the database logic at each stage.

Secondly, our application takes advantage of cloud services. We hosted the application via the LAMP (Linux, Apacahe2, MySQL and PHP) stack on an Ubuntu Virtual Microsoft on Azure. We ran the MySQL server on the same Virtual Machine as the Apache2 server which meant only one Virtual Machine was required. The Apache2 server was configured to execute PHP files at the api/ domain and serve the website files at the root (/) domain. Additionally, we used a deploy script as seen in server/deploy.sh to automate the process of deploying the php files, website files and changes to the stored procedure.

Lastly, we made use of task automation using CRON. To ensure a smooth functioning of the bidding logic, three tasks need to be checked for regularly. These are (1) completing expired auctions and notifying sellers and the highest bidder while inviting both to leave feedback, (2) notifying user with a watch list about new bids for items on these lists, and (3) generating the recommendations.. In order to trigger these tasks via an API endpoint every minute, the server uses *cron*, a Unix application, to send an HTTP request via curl to each of these endpoints.

6 Normalisation Analysis

A major aim of relational database design is to group attributes into relations that minimise data redundancies. By building our database in the third normal form, we ensured less redundancies and thereby a smaller vulnerability to update anomalies, leading to high data integrity.

The first normal form is achieved through three aspects. First, we ensured that every table has a primary or a compound key. Second, repeating groups were avoided by separating any groups related to an entity such as the *Category* and *ItemCategory* table. Third, only atomic values are used, meaning that there are no elements where the data can be further broken down, i.e. *name*, *description* and *bidAmount* are all atomic variables.

The second normal form means that in addition to the first normal form's criteria, all non-key attributes are fully functional dependent on the primary key. If attribute A is dependent on attribute B, but is not dependent on a proper subset of B, then A is considered fully functional dependent on A. If we included the category name into the *ItemCategory* table, it would not have satisfied the second normal form as the variable *name* is dependent on the *categoryID* which is only part of the compound key. By breaking the table into two tables - *ItemCategory* and *Category* - it complies with the second normal form.

A third normal form exists if additionally to the criteria of a second normal form, all non-primary fields are determined only by the primary keys of that relation and not by any non-prime attributes. The dependency of these non-primary fields is called transitive. As we put all key attributes that are functionally dependent on another attribute as primary keys in different tables, our database does not contain any transitive dependencies. It would for example not be in third order if the name of the item was used in the recommendations table as *itemName* is dependent on the *itemID* which is already included in this table.

7 Explanation of Database Queries

All SQL queries used by the application can be found in appendix A with a short explanation for each. As mentioned in section 4, we used stored procedures for all queries, thereby encapsulating their functionality and improving readability as well as modularity. By passing in different parameters, queries are adapted to current needs and function well for different objects. This also helps to protect against common SQL injection attacks and usually results in performance improvements as queries are already stored in executable form and do not have to be interpreted from PHP every time.

Appendix A

Gets the ID of the admin

```
CREATE PROCEDURE p_Admin_sel_user_id
  (p_UserID CHAR(36))
  BEGIN
    SET @UserID = p_UserID;
    SELECT *
    FROM Admin
    WHERE UserID = @UserID;
  END$
```

Deletes the admin based on the user ID

```
CREATE PROCEDURE p_Admin_del_id(p_UserID CHAR(36))
  BEGIN
    SET @UserID = p_UserID;
    DELETE FROM Admin
    WHERE UserID = @UserID;
  END$
```

Creates a new admin based on the user ID

```
CREATE PROCEDURE p_Admin_ins(UserID CHAR(36))
  BEGIN
    SET @UserID = UserID;
    INSERT INTO Admin (UserID, CreatedAt) VALUES (@UserID, NOW());
  END$
```

Returns information about all bids

```
CREATE PROCEDURE p_Bid_sel_all()
  BEGIN SELECT *
        FROM Bid;
  END$
```

Returns all information about the largest bid

```
CREATE PROCEDURE p_Bid_sel_largest(p_ItemID INT(11))
  BEGIN SET @ItemID = p_ItemID;
    SELECT *
    FROM Bid
    WHERE ItemID = @ItemID
    ORDER BY Bid.BidAmount DESC
    LIMIT 1;
  END$
```

Returns the information on one bid based on its ID

```
CREATE PROCEDURE p_Bid_sel_id(p_ID INT(11))
  BEGIN SET @p_ID = p_ID;
    SELECT *
    FROM Bid
    WHERE ID = @p_ID;
  END$
```

Returns all bids from one specific user based on the user ID

```
CREATE PROCEDURE p_Bid_sel_user(p_UserID CHAR(36))
  BEGIN SET @UserID = p_UserID;
    SELECT *
    FROM Bidv
    WHERE UserID = @UserID;
  END$
```

Returns all bids for a specific item based on the item ID

```
  CREATE PROCEDURE p_Bid_sel_item(p_ItemID CHAR(36))
    BEGIN SET @ItemID = p_ItemID;
      SELECT *
      FROM Bid
      WHERE ItemID = @ItemID;
    END$
```

Deletes a bid based on its ID

```
CREATE PROCEDURE p_Bid_del_id(p_ID INT(11))
  BEGIN SET @p_ID = p_ID;
    DELETE FROM Bid
    WHERE ID = @p_ID;
  END$
```

Creates a bid

```
CREATE PROCEDURE p_Bid_ins(ItemID CHAR(36), UserID CHAR(36), BidAmount INT(13))
  BEGIN
    SET @ItemID = ItemID;
    SET @UserID = UserID;
    SET @BidAmount = BidAmount;
    INSERT INTO Bid (ID, ItemID, UserID, BidAmount, CreatedAt)
    VALUES (UUID(), @ItemID, @UserID, @BidAmount, NOW());
  END$
```

Get the distinct emails for those who have bid on the item

```
CREATE PROCEDURE p_Bid_get_bidders_email(ItemID CHAR(36))
  BEGIN
    SET @ItemID = ItemID;
    SELECT User.ID, User.Email FROM Bid
    LEFT JOIN User ON User.ID = Bid.UserID
      WHERE Bid.ItemID = @ItemID
    GROUP BY User.ID;
  END$
```

Lists all categories created

```
CREATE PROCEDURE p_Category_sel_all()
  BEGIN SELECT *
        FROM Category;
  END$
```

Returns a certain category based on its ID

```
CREATE PROCEDURE p_Category_sel_id(p_ID INT(255))
  BEGIN SET @p_ID = p_ID;
    SELECT *
    FROM Category
    WHERE ID = @p_ID;
  END$
```

Creates a new category

```
CREATE PROCEDURE p_Category_ins(Name VARCHAR(255))
  BEGIN SET @Name = Name;
    INSERT INTO Category (Name, CreatedAt) VALUES (@Name, NOW());
  END$
```

Returns all information of items that are in a certain category

```
CREATE PROCEDURE p_Category_sel_items(p_ID INT(255))
  BEGIN SET @p_ID = p_ID;
    SELECT Item.*, CONCAT(User.FirstName, " ", User.LastName) AS SellerName,
feedback.Rating AS SellerRating
    FROM Category
      LEFT JOIN ItemCategory ON Category.ID = ItemCategory.CategoryID
      LEFT JOIN Item ON ItemCategory.ItemID = Item.ID
      LEFT JOIN User ON User.ID = Item.SellerID
      LEFT JOIN (SELECT
                   ToUserID,
                   AVG(Rating) AS Rating
                 FROM Feedback
                 GROUP BY ToUserID) feedback
        ON feedback.ToUserID = Item.SellerID
    WHERE Category.ID = @p_ID;
  END$
```

Returns all feedback given

```
CREATE PROCEDURE p_Feedback_sel_all()
  BEGIN SELECT *
        FROM Feedback;
  END$
```

Returns a specific feedback based on who its from and the item

```
CREATE PROCEDURE p_Feedback_sel(p_FromUserID CHAR(36), p_ItemID CHAR(36))
  BEGIN SET @FromUserID = p_FromUserID;
    SET @ItemID = p_ItemID;
    SELECT *
    FROM Feedback
    WHERE ItemID  = @ItemID AND FromUserID = @FromUserID;
  END$
```

Deletes a feedback based on its ID

```
CREATE PROCEDURE p_Feedback_del_id(p_ID CHAR(36))
  BEGIN SET @ID = p_ID;
    DELETE FROM Feedback
    WHERE ID = @ID;
  END$
```

Creates a new feedback entry

```
CREATE PROCEDURE p_Feedback_ins(ToUserID CHAR(36), FromUserID CHAR(36), ItemID
CHAR(36), Rating TINYINT(4))
  BEGIN SET @ToUserID = ToUserID;
    SET @FromUserID = FromUserID;
    SET @ItemID = ItemID;
    SET @Rating = Rating;
    INSERT INTO Feedback (ToUserID, FromUserID, ItemID, Rating, CreatedAt)
    VALUES (@ToUserID, @FromUserID, @ItemID, @Rating, NOW());
  END$
```

Calculate the average rating of the user

```
CREATE PROCEDURE p_Feedback_calc_user()
  BEGIN
    SELECT ToUserID, AVG(Rating) AS Rating
    FROM Feedback
    GROUP BY ToUserID;
  END$
```

Returns all created items

```
  CREATE PROCEDURE p_Item_sel_all()
    BEGIN SELECT
            Item.ID                                      AS ID,
            Item.Name                                    AS Name,
            Item.Description                             AS Description,
            Item.AuctionStart                            AS AuctionStart,
            Item.AuctionEnd                              AS AuctionEnd,
            Item.AuctionFinished                         AS AuctionFinished,
            Item.StartingPrice                           AS StartingPrice,
            Item.ReservePrice                            AS ReservePrice,
            Item.FinalPrice                              AS FinalPrice,
            Item.PhotoURL                                AS PhotoURL,
            Item.SellerID                                AS SellerID,
            Item.BuyerID                                 AS BuyerID,
            Item.Views                                   AS Views,
            Item.CreatedAt                               AS CreatedAt,
            CONCAT(User.FirstName, ' ', User.LastName) AS SellerName,
            feedback.Rating                              AS SellerRating,
            ItemCategory.CategoryID                      AS CategoryID,
            bid.LargestBid                               AS LargestBid
          FROM Item
            LEFT JOIN User ON Item.SellerID = User.ID
            LEFT JOIN (SELECT
                         ToUserID,
                         AVG(Rating) AS Rating
                       FROM Feedback
                       GROUP BY ToUserID) feedback ON feedback.ToUserID =
Item.SellerID
            LEFT JOIN ItemCategory ON ItemCategory.ItemID = Item.ID
            LEFT JOIN (SELECT
                         Bid.ItemID,
                         MAX(BidAmount) AS LargestBid
                       FROM Bid
                       GROUP BY Bid.ItemID) bid ON bid.ItemID = Item.ID;
    END$
```

Allows to search through all created items based on their name,
their description or their finished status and returns a list of those

```
  CREATE PROCEDURE p_Item_search(p_Query CHAR(40), p_CategoryID INT(255))
    BEGIN SET @query = CONCAT('%', p_Query, '%');
      SET @CategoryID = p_CategoryID;
      SELECT
        Item.ID                                      AS ID,
        Item.Name                                    AS Name,
        Item.Description                             AS Description,
        Item.AuctionStart                            AS AuctionStart,
        Item.AuctionEnd                              AS AuctionEnd,
        Item.AuctionFinished                         AS AuctionFinished,
        Item.StartingPrice                           AS StartingPrice,
        Item.ReservePrice                            AS ReservePrice,
        Item.FinalPrice                              AS FinalPrice,
        Item.PhotoURL                                AS PhotoURL,
        Item.SellerID                                AS SellerID,
        Item.BuyerID                                 AS BuyerID,
        Item.Views                                   AS Views,
        Item.CreatedAt                               AS CreatedAt,
        CONCAT(User.FirstName, ' ', User.LastName) AS SellerName,
        feedback.Rating                              AS SellerRating,
        ItemCategory.CategoryID                      AS CategoryID,
        bid.LargestBid                               AS LargestBid
      FROM Item
        LEFT JOIN User ON Item.SellerID = User.ID
        LEFT JOIN (SELECT
                     ToUserID,
                     AVG(Rating) AS Rating
                   FROM Feedback
                   GROUP BY ToUserID) feedback ON feedback.ToUserID = User.ID
        LEFT JOIN ItemCategory ON ItemCategory.ItemID = Item.ID
        LEFT JOIN (SELECT
                     Bid.ItemID,
                     MAX(BidAmount) AS LargestBid
                   FROM Bid
                   GROUP BY Bid.ItemID) bid ON bid.ItemID = Item.ID
      WHERE Item.Name LIKE @query OR Item.Description LIKE @query
                                     AND Item.AuctionFinished IS NOT NULL
                                     AND (@CategoryID IS NULL OR @CategoryID =
ItemCategory.CategoryID);
    END$
```

Returns a specific item based on its ID

```
CREATE PROCEDURE p_Item_sel_id(p_ID CHAR(36))
  BEGIN SET @p_ID = p_ID;
    SELECT
      Item.ID                                        AS ID,
      Item.Name                                      AS Name,
      Item.Description                               AS Description,
      Item.AuctionStart                              AS AuctionStart,
      Item.AuctionEnd                                AS AuctionEnd,
      Item.AuctionFinished                           AS AuctionFinished,
      Item.StartingPrice                             AS StartingPrice,
      Item.ReservePrice                              AS ReservePrice,
      Item.FinalPrice                                AS FinalPrice,
      Item.PhotoURL                                  AS PhotoURL,
      Item.SellerID                                  AS SellerID,
      Item.BuyerID                                   AS BuyerID,
      Item.Views                                     AS Views,
      Item.CreatedAt                                 AS CreatedAt,
      CONCAT(User.FirstName, ' ', User.LastName) AS SellerName,
      feedback.Rating                                AS SellerRating,
      ItemCategory.CategoryID                        AS CategoryID,
      bid.LargestBid                                 AS LargestBid
    FROM Item
      LEFT JOIN User ON Item.SellerID = User.ID
      LEFT JOIN (SELECT
                   ToUserID,
                   AVG(Rating) AS Rating
                 FROM Feedback
                 GROUP BY ToUserID) feedback ON feedback.ToUserID = User.ID
      LEFT JOIN ItemCategory ON ItemCategory.ItemID = Item.ID
      LEFT JOIN (SELECT
                   Bid.ItemID,
                   MAX(BidAmount) AS LargestBid
                 FROM Bid
                 GROUP BY Bid.ItemID) bid ON bid.ItemID = Item.ID
    WHERE Item.ID = @p_ID;
  END$
```

Deletes an item based on its ID

```
CREATE PROCEDURE p_Item_del_id(p_ID CHAR(36))
  BEGIN SET @ID = p_ID;
    START TRANSACTION;
    DELETE FROM Bid WHERE ItemID = @ID;
    DELETE FROM View WHERE ItemID = @ID;
    DELETE FROM ItemCategory WHERE ItemID = @ID;
    DELETE FROM Feedback WHERE ItemID = @ID;
    DELETE FROM Item WHERE ID = @ID;
    COMMIT;
  END$
```

Creates a new item

```
  CREATE PROCEDURE p_Item_ins(Name         VARCHAR(255), Description TEXT,
                              AuctionStart DATETIME, AuctionEnd DATETIME,
StartingPrice INT(13),
                              ReservePrice INT(13), PhotoURL VARCHAR(255),
                              SellerID     CHAR(36), CategoryID CHAR(36))
    BEGIN
      SET @Name = Name;
      SET @Description = Description;
      SET @AuctionStart = AuctionStart;
      SET @AuctionEnd = AuctionEnd;
      SET @StartingPrice = StartingPrice;
      SET @ReservePrice = ReservePrice;
      SET @PhotoURL = PhotoURL;
      SET @SellerID = SellerID;
      SET @CategoryID = CategoryID;
      SET @UUID = UUID();
      INSERT INTO Item (ID, Name, Description, AuctionStart, AuctionEnd,
StartingPrice, ReservePrice, PhotoURL, SellerID, CreatedAt)
      VALUES
        (@UUID, @Name, @Description, @AuctionStart, @AuctionEnd,
@StartingPrice, @ReservePrice, @PhotoURL, @SellerID,
         NOW());
      IF (CategoryID IS NOT NULL)
      THEN
        CALL p_ItemCategory_ins(@UUID, @CategoryID);
      END IF;
    END$
```

Updates the values of an item based on its ID

```
  CREATE PROCEDURE p_Item_upd(p_ID           CHAR(36), p_Name VARCHAR(255),
p_Description TEXT, p_AuctionStart DATETIME,
                              p_AuctionEnd   DATETIME, p_AuctionFinished
DATETIME, p_StartingPrice INT(13),
                              p_ReservePrice INT(13), p_FinalPrice INT(13),
p_PhotoURL VARCHAR(255),
                              p_SellerID     CHAR(36))
    BEGIN SET @p_ID = p_ID;
      UPDATE Item
      SET ID         = p_ID, Name = p_Name, Description = p_Description,
AuctionStart = p_AuctionStart,
        AuctionEnd   = p_AuctionEnd, AuctionFinished = p_AuctionFinished,
StartingPrice = p_StartingPrice,
        ReservePrice = p_ReservePrice, FinalPrice = p_FinalPrice, PhotoURL =
p_PhotoURL, SellerID = p_SellerID
      WHERE ID = @p_ID;
    END$
```

Increments the view counter of an item by 1

```
CREATE PROCEDURE p_Item_incr_views(p_ID CHAR(36))
  BEGIN SET @p_ID = p_ID;
    UPDATE Item
    SET Views = Views + 1
    WHERE ID = @p_ID;
  END$
```

Returns all items a user sells based on his ID

```
CREATE PROCEDURE p_Item_seller_id(p_SellerID CHAR(36))
  BEGIN SET @SellerID = p_SellerID;
    SELECT *
    FROM Item
    WHERE SellerID = @SellerID;
  END$
```

Returns all items that a user has bid on.

```
  CREATE PROCEDURE p_Items_bidded_on(p_UserID CHAR(36))
    BEGIN SET @UserID = p_UserID;
      SELECT
        Item.ID              AS ID,
        Item.Name            AS Name,
        Item.Description     AS Description,
        Item.AuctionStart    AS AuctionStart,
        Item.AuctionEnd      AS AuctionEnd,
        Item.AuctionFinished AS AuctionFinished,
        Item.StartingPrice   AS StartingPrice,
        Item.ReservePrice    AS ReservePrice,
        Item.FinalPrice      AS FinalPrice,
        Item.PhotoURL        AS PhotoURL,
        Item.SellerID        AS SellerID,
        Item.BuyerID         AS BuyerID,
        Item.Views           AS Views,
        seller.Name          AS SellerName,
        feedback.Rating      AS SellerFeedback,
        bid.BidAmount        AS HighestBid,
        bid.CreatedAt        AS CreatedAt

      FROM Item

        INNER JOIN (SELECT
                      Bid.ItemID,
                      MAX(BidAmount) AS HighestBid
                    FROM Bid
                    WHERE Bid.UserID = @UserID
                    GROUP BY Bid.ItemID) AS maxBid
          ON Item.ID = maxBid.ItemID

        INNER JOIN (SELECT Bid.*
                    FROM Bid
                    WHERE Bid.UserID = @UserID) bid ON bid.BidAmount =
maxBid.HighestBid


        # Get the sellers name
        LEFT JOIN (SELECT
                     User.ID,
                     CONCAT(User.FirstName, ' ', User.LastName) AS Name
                   FROM User) seller
          ON seller.ID = Item.SellerID
        # Get the Sellers rating
        LEFT JOIN (SELECT
                     ToUserID,
                     AVG(Rating) AS Rating
                   FROM Feedback
                   GROUP BY ToUserID) feedback
          ON feedback.ToUserID = Item.SellerID;
    END$
```

Ends an auction once its ending time has passed, checks whether the highest bid is above the reserve price and updates the bid to show its final price and highest bidder. If the highest bid is below the reserve price, the item is not sold.

```
  CREATE PROCEDURE p_Item_end_auctions()
    BEGIN
      SET autocommit = 0;


      CREATE TEMPORARY TABLE EndingAuctions (
        ID          CHAR(36),
        Name        VARCHAR(255),
        PhotoURL    VARCHAR(255),
        SellerID    CHAR(36),
        SellerName VARCHAR(255),
        HighestBid INT(13),
        BuyerID     CHAR(36),
        AuctionFinished DATETIME
      );

      START TRANSACTION;

      INSERT INTO EndingAuctions SELECT
                                    item.ID
AS ID,
                                    item.Name
AS Name,
                                    item.PhotoURL
AS PhotoURL,
                                    item.SellerID
AS SellerID,
                                    CONCAT(User.FirstName, ' ', User.LastName)
AS SellerName,
                                    item.AuctionFinished,
                                    CASE
                                    WHEN bid.highestBid > item.ReservePrice
                                      THEN bid.highestBid
                                    ELSE NULL END
AS 'HighestBid',
                                    CASE WHEN bid.highestBid > item.ReservePrice
                                      THEN bid.UserID
                                    ELSE NULL END
AS 'BuyerID'
                                  FROM Item AS item
                                    LEFT JOIN (
                                                SELECT
                                                  MAX(Bid.BidAmount) AS
highestBid,
                                                  Bid.UserID,
                                                  Bid.ItemID
                                                FROM Bid
```

```
                                        GROUP BY Bid.ItemID
                                      ) bid
                                    ON bid.ItemID = item.ID
                                  LEFT JOIN User ON item.SellerID = User.ID
                                WHERE
                                  item.AuctionFinished IS NULL
                                  AND item.AuctionEnd < now();

    UPDATE Item
      INNER JOIN EndingAuctions
        ON Item.ID = EndingAuctions.ID
    SET
      Item.AuctionFinished = now(),

      Item.FinalPrice      = EndingAuctions.HighestBid,

      Item.BuyerID         = EndingAuctions.BuyerID;

    # Return the finished auctions
    SELECT *
    FROM EndingAuctions;
    COMMIT;
  END$
```

Adds an image to an item

```
  CREATE PROCEDURE p_Item_upl_image(p_ID CHAR(36), p_PhotoURL VARCHAR(255))
    BEGIN SET @p_ID = p_ID, @p_PhotoURL = p_PhotoURL;
      UPDATE Item
      SET PhotoURL = @p_PhotoURL
      WHERE ID = @p_ID;
    END$
```

Returns a list of all items with corresponding category IDs

```
CREATE PROCEDURE p_ItemCategory_sel_all()
  BEGIN SELECT *
        FROM ItemCategory;
  END$
```

Returns an item and its category based on the item and category ID

```
CREATE PROCEDURE p_ItemCategory_sel_id(p_ItemID CHAR(36), p_CategoryID
INT(255))
  BEGIN SET @p_ItemID = p_ItemID;
    SET @p_CategoryID = p_CategoryID;
    SELECT *
    FROM ItemCategory
    WHERE ItemID = @p_ItemID AND CategoryID = @p_CategoryID;
  END$
```

Deletes a category from an item based on the item and category ID

```
CREATE PROCEDURE p_ItemCategory_del_id(p_ItemID CHAR(36), p_CategoryID
INT(255))
  BEGIN
    DELETE FROM ItemCategory
    WHERE ItemID = p_ItemID AND CategoryID = p_CategoryID;
  END$
```

Assigns an item to a category

```
CREATE PROCEDURE p_ItemCategory_ins(p_ItemID CHAR(36), p_CategoryID INT(255))
  BEGIN
    INSERT INTO ItemCategory (ItemID, CategoryID, CreatedAt)
    VALUES
      (p_ItemID, p_CategoryID, NOW());
  END$
```

Returns a list of all recommended items and users they are recommended to

```
CREATE PROCEDURE p_Recommendation_sel_all()
  BEGIN SELECT *
        FROM Recommendation;
  END$
```

Returns IDs of recommended items for a user based on viewed items

```
CREATE PROCEDURE p_Recommendation_sel_id(p_UserID CHAR(36), p_ItemID CHAR(36))
  BEGIN SET @p_UserID = p_UserID;
    SET @p_ItemID = p_ItemID;
    SELECT *
    FROM Recommendation
    WHERE UserID = @p_UserID AND ItemID = @p_ItemID;
  END$
```

Deletes item recommendation based on the ID of the user and the item

```
CREATE PROCEDURE p_Recommendation_del_id(p_UserID CHAR(36), p_ItemID CHAR(36))
  BEGIN SET @p_UserID = p_UserID;
    SET @p_ItemID = p_ItemID;
    DELETE FROM Recommendation
    WHERE UserID = @p_UserID AND ItemID = @p_ItemID;
  END$
```

Creates an item recommendation for a user

```
CREATE PROCEDURE p_Recommendation_ins(IN p_UserID CHAR(36),
                                      IN p_ItemID CHAR(36))
  BEGIN
    INSERT INTO Recommendation (UserID,
                                ItemID, CreatedAt)
    VALUES
      (p_UserID,
       p_ItemID, NOW());
  END$
```

Returns a list of all registered users

```
CREATE PROCEDURE p_User_sel_all()
  BEGIN SELECT *
        FROM User;
  END$
```

Returns all information of a user based on his ID

```
CREATE PROCEDURE p_User_sel_id(IN p_ID CHAR(36))
  BEGIN SET @p_ID = p_ID;
    SELECT
      ID,
      FirstName,
      LastName,
      Email
    FROM User
    WHERE ID = @p_ID;
  END$
```

Deletes a user based on his ID

```
CREATE PROCEDURE p_User_del_id(p_ID CHAR(36))
  BEGIN SET @p_ID = p_ID;
    DELETE FROM User
    WHERE ID = @p_ID;
  END$
```

Creates a new user

```
CREATE PROCEDURE p_User_ins(p_FirstName VARCHAR(255),
                            p_LastName  VARCHAR(255),
                            p_Email     VARCHAR(255),
                            p_Password  VARCHAR(255)
)
  BEGIN
    INSERT INTO User (ID, FirstName, LastName, Email, Password, CreatedAt)
    VALUES (UUID(), p_FirstName, p_LastName, p_Email, p_Password, NOW());
  END$
```

Updates a user's information based on his ID

```
CREATE PROCEDURE p_User_upd(p_ID        CHAR(36),
                            p_FirstName VARCHAR(255),
                            p_LastName  VARCHAR(255),
                            p_Email     VARCHAR(255))
  BEGIN
    UPDATE User
    SET FirstName = p_FirstName,
      LastName    = p_LastName,
      Email       = p_Email
    WHERE ID = p_ID;
  END$
```

Returns a user based on his password and email

```
  CREATE PROCEDURE p_User_login(p_Email VARCHAR(255), p_PasswordHash
VARCHAR(255))
    BEGIN SET @p_PasswordHash = p_PasswordHash;
      SET @p_Email = p_Email;
      SELECT *
      FROM User
      WHERE Password = @p_PasswordHash AND Email = @p_Email;
    END$
```

Views an item

```
CREATE PROCEDURE p_View(p_UserID VARCHAR(255),
                        p_ItemID VARCHAR(255))
  BEGIN SET @UserID = p_UserID;
    SET @ItemID = p_ItemID;
    START TRANSACTION;
    IF NOT EXISTS(SELECT *
                  FROM View
                  WHERE UserID = @UserID AND ItemID = @ItemID)
    THEN
      INSERT INTO View (UserID, ItemID, Count, CreatedAt) VALUES (@UserID,
@ItemID, 0, NOW());
    END IF;

    UPDATE View
    SET Count = Count + 1
    WHERE UserID = @UserID AND ItemID = @ItemID;

    COMMIT;

  END$
```

Creates the table for collobrative filtering

```
CREATE PROCEDURE p_View_init_filtering()
  BEGIN
    DELETE FROM oso_user_ratings;
    INSERT INTO oso_user_ratings (user_id, item_id, rating)
      SELECT
        View.UserID,
        View.ItemID,
        View.Count / maxView.MaxView
      FROM View
        LEFT JOIN (SELECT
                     View.UserID,
                     MAX(Count) AS MaxView
                   FROM View
                   GROUP BY View.UserID) maxView ON View.UserID =
maxView.UserId;

    COMMIT;

  END$
```

Returns an item from the watch list based on the user and item ID

```
  CREATE PROCEDURE p_Watch_sel_id(p_UserID CHAR(36), p_ItemID CHAR(36))
    BEGIN SET @p_UserID = p_UserID, @p_ItemID = p_ItemID;
      SELECT *
      FROM Watch
      WHERE UserID = @p_UserID AND ItemID = @p_ItemID;
    END$
```

Deletes an item from the user's watch list based on the user and item ID

```
  CREATE PROCEDURE p_Watch_del_id(p_UserID CHAR(36), p_ItemID CHAR(36))
    BEGIN SET @UserID = p_UserID, @ItemID = p_ItemID;
      DELETE FROM Watch
      WHERE UserID = @UserID AND ItemID = @ItemID;
    END$
```

Adds an item on the user's watch list

```
  CREATE PROCEDURE p_Watch_ins(p_UserID CHAR(36), p_ItemID CHAR(36))
    BEGIN
      SET @UserID = p_UserID, @ItemID = p_ItemID;
      INSERT INTO Watch (UserID, ItemID, CreatedAt) VALUES (@UserID, @ItemID,
NOW());
    END$
```

Returns all items from the user's watch list based on the user's ID

```
CREATE PROCEDURE p_Watch_sel_all_userid(p_UserID CHAR(36))
  BEGIN SET @UserID = p_UserID;
    SELECT *
    FROM Watch
    WHERE UserID = @UserID;
  END$
```

Returns all users that have a specific item on their watch lists

```
CREATE PROCEDURE p_Watch_sel_all_from_item(p_ItemID CHAR(36))
  BEGIN SET @ItemID = p_ItemID;
    SELECT UserID
    FROM Watch
    WHERE ItemID = @ItemID;
  END$
```