Corporate State University Baden-Wüerttemberg Mannheim

Database Project

BidBits

Study program Business Informatics

Spezialisation Data Science

Authors:	David Schäfer, Eric Echtermeyer, An-Phi Dang
Student numbers:	7086451, 6373947, 7558992
Course:	WWI-21-DSA
Director of studies:	Prof. Dr.-Ing. habil. Dennis Pfisterer
Lecturer:	Petko Rutesic

Contents

1 Database specification

In the subsequent section, the specifics of the database will be explained. Figure 1.1 showcases the Entity-Relationship Diagram (ERD) for the database schema which visually represents the structure of the implemented database. In the following, detailed definitions for used tables, relationships, views and stored procedures will be explained too.

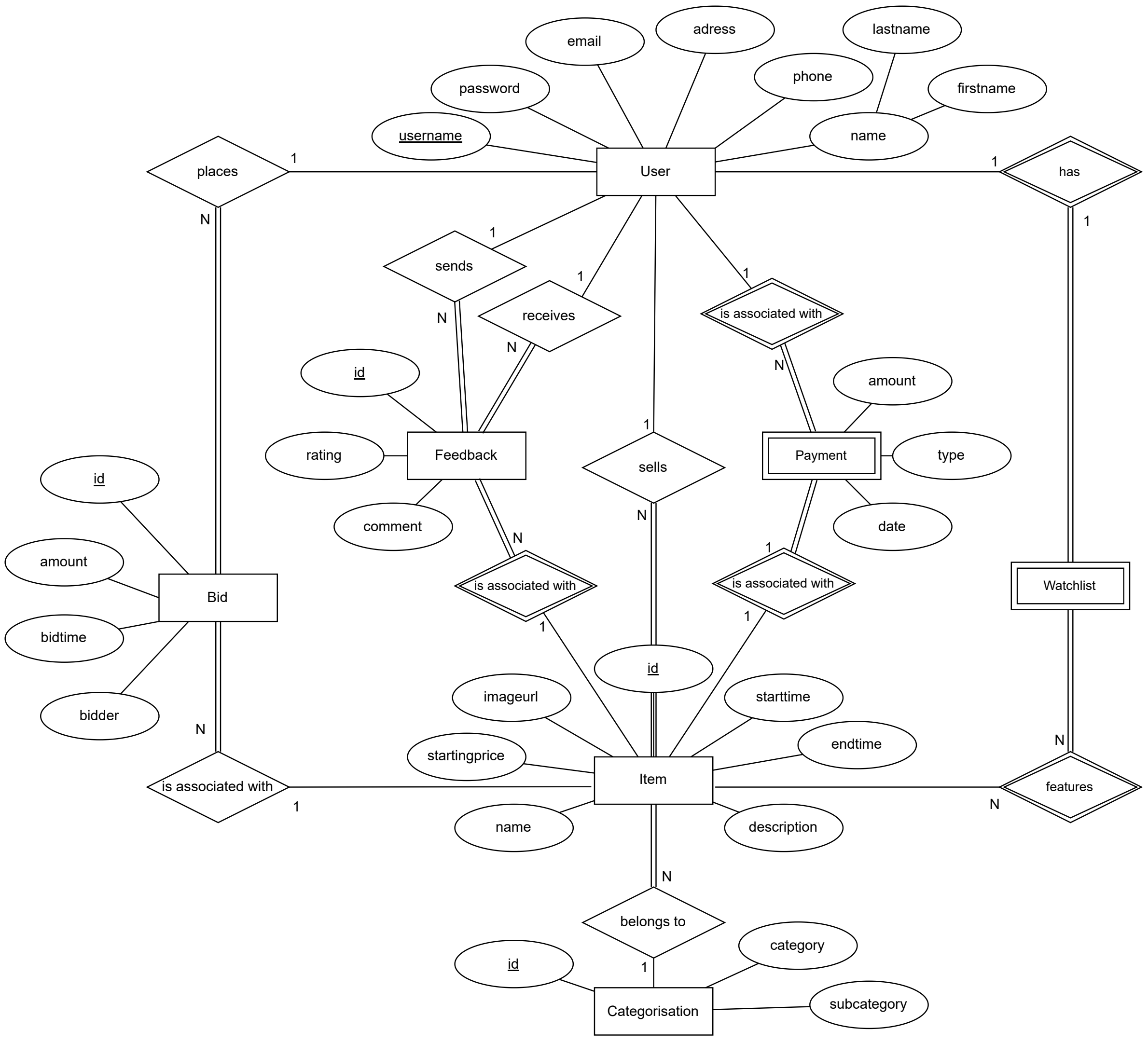

Figure 1.1: Entity-Relationship Diagram

1.1 Tables

The user table is designed to store all details about the users of the auction system. This includes columns such as 'username', which serves as the primary key and a unique identifier for each user, 'email', the user's unique email address, 'password', the user's hashed password for enhanced security, 'firstName' and 'lastName', which are the first and last name of the user respectively, 'name', a composite of the first and last names, 'address', the user's physical address, and 'phone', the unique phone number of the user. All attributes must have values to create a new user entry.

```
        CREATE TABLE "user" (
                username VARCHAR(255) PRIMARY KEY,
                email EMAIL NOT NULL UNIQUE,
                password VARCHAR(255) NOT NULL,
                firstName VARCHAR(50) NOT NULL,
                lastName VARCHAR(50) NOT NULL,
                name VARCHAR(100) GENERATED ALWAYS AS (firstName || '␣' || lastName) STORED,
                address TEXT NOT NULL,
                phone VARCHAR(20) NOT NULL UNIQUE
        );
```

The item table is responsible for tracking all items listed for auction. It comprises columns such as 'id', a unique identifier for each item and the primary key for this table, 'name', the item's name, 'description', a detailed description of the item, 'startingPrice', the initial price for the auction of this item, 'startTime' and 'endTime', the timestamps for the beginning and end of the auction, 'imageUrl', the URL for an image of the item, 'user_username', the username of the user who listed the item, a foreign key referencing the user table, and 'category_id', the id of the item's category, a foreign key referencing the categorisation table.

```
        CREATE TABLE Item (
                id INTEGER DEFAULT nextval('item_sequence') PRIMARY KEY,
                name VARCHAR(50) NOT NULL,
                description TEXT NOT NULL,
                startingPrice NUMERIC DEFAULT 0,
                startTime TIMESTAMP NOT NULL,
                endTime TIMESTAMP NOT NULL,
                imageUrl TEXT NOT NULL,
                user_username VARCHAR(50) REFERENCES "user"(username)
        ON DELETE SET NULL,
                category_id INTEGER REFERENCES Categorisation(id)
        );
```

The categorisation table categorizes the items being auctioned. It consists of three columns - 'id', 'category', and 'subcategory'. 'id' is a unique identifier for each category or subcategory pairing and serves as the primary key for this table. 'category' and 'subcategory' are text-based columns that describe the item category and the corresponding subcategory.

```
        CREATE TABLE Categorisation (
                id SERIAL PRIMARY KEY,
                category VARCHAR(50) NOT NULL,
                subcategory VARCHAR(50) NOT NULL
        );
```

The bid table contains information regarding all bids placed on the items. It includes 'id', a unique identifier for each bid and the primary key for this table, 'amount', the bid's amount, 'bidTime', the timestamp when the bid was placed, 'user_username', the username of the user who placed the bid, a foreign key referencing the user table, and 'item_id', the id of the item being bid on, a foreign key referencing the item table.

```
        CREATE TABLE Bid (
                id SERIAL PRIMARY KEY,
                amount NUMERIC NOT NULL,
                bidTime TIMESTAMP NOT NULL,
                user_username VARCHAR(50) REFERENCES "user"(username)
        ON DELETE SET NULL,
                item_id INTEGER REFERENCES Item(id)
        );
```

The watchlist table allows users to track items they are interested in. It includes 'user_username', the username of the user interested in an item, a foreign key referencing the user table, and 'item_id', the id of the item the user is interested in, a foreign key referencing the item table. The combination of 'user_username' and 'item_id' is unique, preventing duplicate entries in the watchlist.

```
        CREATE TABLE Watchlist (
                user_username VARCHAR(50) REFERENCES "user"(username)
        ON DELETE CASCADE,
                item_id SERIAL REFERENCES Item(id),
        CONSTRAINT unique_watchlist_entry UNIQUE (user_username,
            ↪ item_id)
        );
```

The feedback table facilitates users in leaving feedback about their transactions. It comprises 'feedbackID', a unique identifier for each feedback and the primary key for this table, 'item-id', the id of the item which was sold and to which the feedback is refering, 'rating', the rating given by the sender to the receiver on a scale from 0 to 10, 'comment', a text field for additional comments, 'sender' and 'receiver', the usernames of the user giving and receiving the feedback, both foreign keys referencing the user table.

```
        CREATE TABLE Feedback (
                feedbackID SERIAL PRIMARY KEY,
                item_id INTEGER REFERENCES Item(id),
                rating INTEGER NOT NULL CHECK (rating BETWEEN 0 AND 10),
                comment TEXT,
                sender VARCHAR(50) REFERENCES "user"(username)
        ON DELETE SET NULL,
                receiver VARCHAR(50) REFERENCES "user"(username)
        ON DELETE SET NULL
        );
```

The payment table records all payments made for the items. It includes 'amount', the amount paid, 'date', the timestamp when the payment was made, 'type', the method of payment, 'user_username', the username of the user making the payment, a foreign key referencing the user table, and 'item_id', the id of the item for which the payment was made, a foreign key referencing the item table.

```
        CREATE TABLE Payment (
                amount NUMERIC NOT NULL,
                date TIMESTAMP NOT NULL,
                type PAYMENT_TYPE NOT NULL,
                user_username VARCHAR(50) REFERENCES "user"(username)
        ON DELETE SET NULL,
                item_id INTEGER REFERENCES Item(id)
        );
```

1.2 Relationships

User Table - Item Table: Each user can list many items for auction, but each item is listed by only one user. The relationship is established via the user_username foreign key in the Item table.

User Table - Bid Table: Each user can place many bids, but each bid is placed by only one user. This relationship is established via the user_username foreign key in the Bid table.

User Table - Watchlist Table: Each user can have many items on their watchlist, but each watchlist entry is linked to only one user. This relationship is established via the user_username foreign key in the Watchlist table.

User Table - Feedback Table: Each user can both give and receive feedback many times, but each feedback entry is given by and given to only one user. This relationship is established via the sender and receiver foreign keys in the Feedback table.

User Table - Payment Table: Each user can make many payments, but each payment is made by only one user. This relationship is established via the user_username foreign key in the Payment table.

Item Table - Bid Table: Each item can have many bids, but each bid is placed on only one item. The relationship is established via the item_id foreign key in the Bid table.

Item Table - Watchlist Table: Each item can be on the watchlist of many users, but each watchlist entry is for only one item. This relationship is established via the item_id foreign key in the Watchlist table.

Item Table - Payment Table: Each item has one payment associated with it, and each payment is linked to only one item. This relationship is established via the item_id foreign key in the Payment table.

Categorisation Table - Item Table: Each category can have many items, but each item belongs to only one category. The relationship is established through the category_id foreign key in the Item table.

1.3 Views

The database also consists of two views, item_status and user_statistics. The items_status view is created to show the current status of each item in the auction. It contains various fields that provide important information about the item, its current highest bid, and the related users (highest bidder and seller). The purpose of this view is to aggregate relevant data from multiple tables and present it in a more readable and understandable format. It can be used to display the current state of items in a user-friendly way on an application or website.

The view is created by joining the item table with a subquery that calculates the highest bid for each item. The result is then joined with the bid table to retrieve the highest bidder's username. The selected fields in this view include the item's name, id, description, image URL, remaining time until the auction ends, the highest bid, the highest bidder's username, and the seller's username.

```sql
CREATE VIEW items_status AS
SELECT
        item.name AS title,
        item.id AS item_id,
        item.description,
        item.imageUrl AS image_path,
        EXTRACT(DAY FROM AGE(item.endtime, CURRENT_TIMESTAMP)) AS time_left,
        max_bids.highest_bid,
        bid.user_username AS highest_bidder,
        item.user_username AS seller
FROM item
JOIN (SELECT item_id, MAX(amount) AS highest_bid FROM bid GROUP BY item_id) AS max_bids
ON item.id = max_bids.item_id
JOIN bid
ON bid.item_id = max_bids.item_id
AND bid.amount = max_bids.highest_bid;
```

The user_statistics materialized view in the database provides an aggregated summary of user activities on the auction platform. Being a materialized view, it stores the result of a complex and potentially time-consuming query, which can be refreshed as required, leading to faster data retrieval.

Each row in this view represents a unique user on the platform. The view contains a variety of fields calculated from different tables to give a comprehensive overview of each user's auction activities. The participated_auctions field shows the total number of unique auctions where the user has placed a bid, as determined from the bid table. Another field, won_auctions, displays the number of auctions that the user has won. It's calculated from the bid table, where the user's bid was the highest and the auction time had ended.

The average_rating field indicates the mean feedback rating received by the user. It's derived from the feedback table by averaging all the ratings received by a user. To understand the financial aspect, two more fields are included: total_expenses and total_income. total_expenses shows the sum of all the payments made by a user for won auctions, calculated from the payment table. On the other hand, total_income calculates the total amount received by a user for the items they have sold. It's calculated by joining the payment and item tables and summing up the amounts where the user is the seller.

```sql
CREATE MATERIALIZED VIEW user_statistics AS
SELECT
    u.username,
    COALESCE(participated_auctions.participated_auctions, 0) AS participated_auctions,
    COALESCE(won_auctions.won_auctions, 0) AS won_auctions,
    COALESCE(CAST(average_feedback.average_rating AS INTEGER), 0) AS average_rating,
    COALESCE(total_expenses.total_expenses, 0) AS total_expenses,
    COALESCE(total_income.total_income, 0) AS total_income
FROM "user" u
LEFT JOIN (
SELECT user_username, COUNT(DISTINCT item_id) AS participated_auctions
FROM bid
GROUP BY user_username
) AS participated_auctions ON u.username = participated_auctions.user_username
LEFT JOIN (
SELECT user_username, COUNT(DISTINCT bid.item_id) AS won_auctions
FROM bid
JOIN items_status ON bid.item_id = items_status.item_id AND bid.amount = items_status.highest_bid
WHERE items_status.time_left < 0
GROUP BY user_username
) AS won_auctions ON u.username = won_auctions.user_username
```

```
        LEFT JOIN (
        SELECT receiver, AVG(rating) AS average_rating
        FROM feedback
        GROUP BY receiver
        ) AS average_feedback ON u.username = average_feedback.receiver
        LEFT JOIN (
        SELECT user_username AS buyer, SUM(amount) AS total_expenses
        FROM payment
        GROUP BY user_username
        ) AS total_expenses ON u.username = total_expenses.buyer
        LEFT JOIN (
        SELECT item.user_username AS seller, SUM(amount) AS
            ↪ total_income
        FROM payment
        JOIN item ON item.id = payment.item_id
        GROUP BY item.user_username
        ) AS total_income ON u.username = total_income.seller;
```

1.4 Stored Procedure

The purpose of the implemented stored procedure is to add a random item to the watchlist of a user when the user account is created. The item must be currently on auction, meaning its end time must be in the future.

A variable random_item_id of type integer is declared to store the id of a randomly selected item. It executes a SELECT query to get the id of a random item that is currently on auction. This is done by ordering the items from the items_status view (which contains the items with their current status) randomly and selecting the first one. Once it has the id of a random item, it inserts a new row into the watchlist table with the username of the new user and the id of the randomly selected item.

This stored procedure is run by a trigger named user_created_trigger. A trigger is a special type of stored procedure that runs automatically when an event associated with a table occurs such as insert, update, or delete. In this case, the trigger is configured to run the add_random_item_to_watchlist() stored procedure each time a new row is inserted into the user table. This is specified by the AFTER INSERT ON "user" part of the trigger creation statement.

```
CREATE OR REPLACE FUNCTION add_random_item_to_watchlist()
RETURNS TRIGGER AS $$
DECLARE
        random_item_id INTEGER;
BEGIN

-- Get the id of a random item which is currently on auction
SELECT item_id
FROM items_status
WHERE time_left > 0
ORDER BY RANDOM()
LIMIT 1
INTO random_item_id;

-- Put that item on the watchlist of the newly created user
INSERT INTO watchlist (user_username, item_id)
VALUES (NEW.username, random_item_id);

RETURN NEW;
END;

$$ LANGUAGE plpgsql;

-- Create a trigger that executes the stored procedure
   ↪ whenever a new entry is added to the user table
CREATE TRIGGER user_created_trigger
AFTER INSERT ON "user"
FOR EACH ROW
EXECUTE FUNCTION add_random_item_to_watchlist();
```

2 Normalization Analysis

Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normal forms are a set of conditions or rules that a relation (table) in a database must adhere to, to qualify as a 'good' structure. This analysis will check whether the relations in the database meet at least the Third Normal Form (3NF).

Normal Form	**Definition**
First Normal Form (1NF)	A relation is in 1NF if it contains an atomic value for each attribute (column) in a record (row). It should also have a primary key that uniquely identifies each record.
Second Normal Form (2NF)	A relation is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key. This essentially means there is no partial dependency of any column on the primary key.
Third Normal Form (3NF)	A relation is in 3NF if it is in 2NF and no non-key attribute is transitively dependent on the primary key.

Table 2.1: Definitions of Normal Forms

1. Categorisation Table The Categorisation table consists of three attributes: id, category, and subcategory. The primary key is 'id', and every 'id' refers to a unique category and subcategory. There are no duplicate or redundant data, and every attribute is atomic, which satisfies 1NF. Since there's only one candidate key (id), and all non-key attributes (category and subcategory) are fully dependent on it, 2NF is satisfied. There's no transitive dependency as there's only one candidate key, and thus the table also satisfies 3NF.

2. User Table The 'user' table has eight attributes: username, email, password, firstName, lastName, name, address, and phone. The primary key is 'username', and every 'username' refers to a unique user record. However, the 'name' attribute is a derived attribute (it's generated always as the concatenation of 'firstName' and 'lastName'). This breaks the rule of 1NF as it introduces redundancy. To satisfy 1NF, we could remove the 'name' attribute and derive it in our queries when needed. Assuming we consider the 'name' attribute as not violating 1NF, the table would meet the conditions for 2NF as there's only one candidate key, and all non-key attributes are fully dependent on it. For

3NF, the table doesn't have any non-key attribute that is transitively dependent on the primary key, so it satisfies 3NF as well.

3. Item Table The Item table has nine attributes: id, name, description, startingPrice, startTime, endTime, imageUrl, user_username, and category_id. The primary key is 'id', and every 'id' refers to a unique item. The table is in 1NF as all attributes are atomic, and each record is unique. The table is in 2NF as all non-key attributes are fully dependent on the primary key. The table is in 3NF as there are no transitive dependencies.

4. Bid Table The Bid table is in 1NF, 2NF, and 3NF. All attributes are atomic, each record is uniquely identified by 'id', all non-key attributes are fully dependent on the primary key, and there are no transitive dependencies.

5. Watchlist Table The Watchlist table has two attributes: user_username and item_id. The primary key is a combination of 'user_username' and 'item_id', and every combination refers to a unique watchlist entry. The table is in 1NF as all attributes are atomic, and each record is unique. The table is in 2NF as there are no non-key attributes, so the condition of full functional dependency is trivially satisfied. The table is also in 3NF as there are no transitive dependencies.

All tables in the given SQL code are in the Third Normal Form (3NF), assuming that the 'name' attribute in the 'user' table doesn't violate the 1NF. If it does, the 'user' table should be modified by removing the 'name' attribute to meet 1NF, 2NF, and 3NF.

3 SQL Queries and Indices

The fields 'item_id' and 'amount' where indexed in order to significantly improve the performance of queries that involve filtering, searching, or joining based on these columns.

```
        CREATE INDEX index_bid_item_id ON Bid (item_id);
        CREATE INDEX index_bid_amount ON Bid (amount);
```

This increases efficiency considerably since those columns used to create the database view 'items_status', which in turn is accessed in 5 separate queries, including those that are the used most frequently. The indexing significantly speeds up data retrieval from these tables which results in faster response times of the website as a whole. In total our web application uses 22 separate queries which were implemented using psycopg2, 7 of which will be explained in this chapter.

```
SELECT
        items_status.*,
        CASE
                WHEN filtered_watchlist.user_username is NUll THEN 0
                ELSE 1
        END as is_watchlist
FROM items_status
        LEFT JOIN (
                SELECT * FROM watchlist WHERE user_username = '{self.
                    ↪ __current_user}'
        ) AS filtered_watchlist ON items_status.item_id =
            ↪ filtered_watchlist.item_id
WHERE items_status.time_left > 0;
```

This nested query retrieves active items from the items_status view and adds an additional column that indicates whether the item is in the users watchlist or not.

```
        REFRESH MATERIALIZED VIEW user_statistics;
        SELECT * FROM user_statistics WHERE username = '{self.
            ↪ __current_user}';
```

This query refreshes the materialized view user_statistics and retrieves statistics data such as the number of won auctions or the total income generated via sold items for the current user.

```sql
SELECT DISTINCT
        items_status.item_id,
        title,
        description,
        image_path,
        highest_bid AS amount,
        CASE
                WHEN highest_bidder = '{self.__current_user}' THEN 'buyer'
                WHEN seller = '{self.__current_user}' THEN 'seller'
        END AS role,
        CASE
                WHEN feedback.rating IS NULL THEN 0
                ELSE 1
        END AS has_feedback
FROM items_status
        LEFT JOIN feedback ON items_status.item_id = feedback.item_id
WHERE time_left <= 0 AND (highest_bidder = '{self.__current_user}' OR seller = '{self.__current_user}');
```

This query retrieves the completed auctions in which the current user participated either as a buyer or a seller. It includes information such as item ID, title, description, image path, highest bid amount and the user's role. The field 'has_feedback' contains the information wether the user already send feedback to the other user which won or sold the item. The frontend uses this information to either display or hide the possibility to give feedback.

```sql
UPDATE \"user\" SET {filled_inputs} WHERE username = '{self.__current_user}';
```

This query updates the user's personal data in the user table. It takes the email, first name, last name, address, and phone as input values and updates the corresponding columns for the current user.

```sql
DELETE FROM watchlist WHERE user_username = '{self.__current_user}'
    ↪ AND item_id = '{item_id}'
```

This query deletes the user's data from the user table. Other tables which reference the user table utilize 'ON DELETE SET NULL'. This way the statistics and auction histories of other users are not impaired once a user is deleted.

```sql
SELECT
        highest_bid AS amount,
        item.endtime AS date,
        CASE FLOOR(RANDOM() * 3)
                WHEN 0 THEN 'Cash'
                WHEN 1 THEN 'Credit␣Card'
                WHEN 2 THEN 'Paypal'
        END AS type,
        items_status.highest_bidder AS user_username,
        items_status.item_id
FROM items_status
        LEFT JOIN payment ON items_status.item_id = payment.item_id
        JOIN item ON items_status.item_id = item.id
WHERE time_left <= 0 AND payment.amount IS NULL AND highest_bid > 0;

INSERT INTO payment VALUES ({auction["amount"]}, '{auction["date"]}',
    ↪ '{auction["type"]}', '{auction["user_username"]}', {auction["
    ↪ item_id"]});
```

This query retrieves information about auctions that have ended but do not have a corresponding payment yet. It then chooses a payment type at random and inserts a corresponding entry in the 'payment' table. It was implemented for demonstration purposes and simulates a direct debit collection for an item whose auction has endet. Because the scope of this university project does not allow hosting a server which automatically executes this query at the very second that the auction has endet, we had to come up with a different solution. That's why this query is triggered each time any other query is executed. This way it is ensured that the user immediately sees updated informations (e.g. user statistics or payment info) once an auction has endet and he interacted with the website. Again this is just for demonstration purposes and we are well aware that this would not be a feasable solution in a real world scenario.