Demo Video: https://www.youtube.com/watch?v=D5bSWbO2ddk
Website URL: https://car-tinder-476522.web.app/

Intro: Car Tinder is an interactive mobile/web application designed to simplify the used car buying experience. Buying a car can be overwhelming due to fluctuating prices, inconsistent vehicle histories, varying fuel economy, and reliability concerns. Our app transforms the process into a fun and engaging experience by allowing users to swipe through personalized car recommendations based on their preference. It's the same high stakes and high financial implications as Tinder, but for cars!

During the development of our project, our implementation stayed very consistent and true to what we discussed in our proposal. We created user profiles, a one-by-one showing of car listings which could be liked or passed on, a history of past interactions, and preferences/filtering. All of these features were mentioned in our proposal. Although we didn't stray from our proposal much, we made small changes, such as physical "like" and "dislike" buttons instead of the Tinder-like swiping feature. This made more sense for our web deployment.

We believe that our project successfully achieved an MVP of a very useful application. We say this because the ease of use is apparent compared to the regular method of car shopping, however, if we were to fully deploy our app, we would need to clean up the front-end and workflows. While testing, we realized that compared to regular car shopping, our app is more fun, engaging, and simple. We successfully gamified car shopping. One of our group members is actually actively used car shopping and mentioned how much nicer our project is to use than standard listing websites.

We did not change the source of our data, however we did end up updating our schema to handle new functionality. Users can already inspect and remove likes individually, so we added a "Clear All Likes" button to reset their history in one step, in case the user wants to start fresh. In order to preserve the history of what a user swiped on we maintain an archive of their deleted swipes, and log how many likes were deleted. To facilitate this we added two tables not in our original UML diagram:

```
CREATE TABLE SwipeArchive (
    archive_id   INT AUTO_INCREMENT PRIMARY KEY,
    user_id      INT NOT NULL,
    listing_id   INT NOT NULL,
    action       ENUM('LIKE', 'PASS') NOT NULL,
    created_at   TIMESTAMP NOT NULL,
    archived_at  TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE UserActionLog (
    log_id          INT AUTO_INCREMENT PRIMARY KEY,
```

```
    user_id          INT NOT NULL,
    removed_likes    INT NOT NULL,
    removed_passes   INT NOT NULL,
    removed_listings INT NOT NULL,
    logged_at        TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

This design is more suitable than our original because it can handle our new features. Without these changes, we would not be able to do the clear all likes function efficiently.

Something that we added to our app was a keyword search feature to find a specific type of car. We realized that although the “tinder-like” one at a time feature is fun, if a user is looking for a very specific car, it may take forever for them to find it. This is why we added a specific search feature. We also added pre-made filters, such as the top 10 liked cars, so users can see what is popular among other users, and a filter for cars where the average price is under $20k, so that users can find models that are affordable to them. Unfortunately, due to our data, we had to make the experience slightly less personal however. For our preferences, we removed fuel type, and transmission preferences because of a lack of data. We also removed the data visualization on the car cards we initially planned on doing due to the same reason.

Our advanced database programs helped with filtering based on what the user wanted to see, and helping us give recommendations.

```
const sql = `
    SELECT
        c.car_id,
        c.make,
        c.model,
        c.year,
        c.mpg,
        ci.image_url,
        u.listing_id,
        u.price,
        avg_data.avg_price
    FROM (
        SELECT
            car_id,
            AVG(price) AS avg_price
        FROM UsedCarListing
        GROUP BY car_id
        HAVING AVG(price) < 20000
    ) AS avg_data
    JOIN Car c ON avg_data.car_id = c.car_id
    JOIN UsedCarListing u ON u.car_id = c.car_id
    LEFT JOIN CarImage ci ON c.car_id = ci.car_id
    ORDER BY avg_data.avg_price ASC;
```

This command calculates the average price for each model and filters the results using the “having” clause so that only cars with an average price under $20,000 are shown. This is helpful

because users who don't necessarily know what cars they are looking for can see common affordable cars and learn what they like.

```
const sql = `
        SELECT
            c.car_id,
            c.make,
            c.model,
            c.year,
            c.mpg,
            ci.image_url,
            u.listing_id,
            u.price,
            COUNT(s.swipe_id) AS total_likes
        FROM Swipe s
        JOIN UsedCarListing u ON s.listing_id = u.listing_id
        JOIN Car c ON u.car_id = c.car_id
        LEFT JOIN CarImage ci ON c.car_id = ci.car_id
        WHERE s.action = 'LIKE'
        GROUP BY
            c.car_id,
            c.make,
            c.model,
            c.year,
            ci.image_url,
            u.listing_id,
            u.price
        ORDER BY total_likes DESC
        LIMIT 10;
```

This command gets the top 10 most liked cars across all users. This acts as a helpful filter as it gives insights into what cars are popular so that someone who doesn't have much car knowledge can still figure out when a listing is good.

Stored procedure:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS sp_get_recommended_cars$$

CREATE PROCEDURE sp_get_recommended_cars(
    IN p_user_id INT,
    IN p_limit INT
)
BEGIN
    DECLARE liked_count INT;
    DECLARE remaining INT;

    SELECT COUNT(*)
    INTO liked_count
    FROM Swipe
    WHERE user_id = p_user_id
      AND action = 'LIKE';

    DROP TEMPORARY TABLE IF EXISTS tmp_recommendations;

    CREATE TEMPORARY TABLE tmp_recommendations (
        car_id INT,
        listing_id INT,
        make VARCHAR(100),
        model VARCHAR(100),
        year INT,
        price DECIMAL(10,2),
        image_url VARCHAR(500),
        mpg INT
    );

    IF liked_count > 0 THEN
```

```
    DROP TEMPORARY TABLE IF EXISTS tmp_user_likes;

    CREATE TEMPORARY TABLE tmp_user_likes (
        make VARCHAR(100),
        model VARCHAR(100),
        like_count INT,
        PRIMARY KEY(make, model)
    );

    INSERT INTO tmp_user_likes (make, model, like_count)
    SELECT
        c.make,
        c.model,
        COUNT(*)
    FROM Swipe s
    JOIN UsedCarListing u ON s.listing_id = u.listing_id
    JOIN Car c ON u.car_id = c.car_id
    WHERE s.user_id = p_user_id
      AND s.action = 'LIKE'
    GROUP BY c.make, c.model;

    INSERT INTO tmp_recommendations
    SELECT
        c.car_id,
        u.listing_id,
        c.make,
        c.model,
        c.year,
        u.price,
        i.image_url,
        c.mpg
    FROM Car c
    JOIN UsedCarListing u ON u.car_id = c.car_id
    LEFT JOIN CarImage i ON i.car_id = c.car_id
    JOIN tmp_user_likes t
      ON t.make = c.make
     AND t.model = c.model
    LEFT JOIN Swipe s
      ON s.user_id = p_user_id
     AND s.listing_id = u.listing_id
    WHERE s.listing_id IS NULL
    ORDER BY
        t.like_count DESC,
        c.year DESC,
        u.price ASC
    LIMIT p_limit;

    DROP TEMPORARY TABLE IF EXISTS tmp_user_likes;
END IF;

SET remaining = p_limit - (SELECT COUNT(*) FROM tmp_recommendations);

IF remaining > 0 THEN

    DROP TEMPORARY TABLE IF EXISTS tmp_existing_recs;

    CREATE TEMPORARY TABLE tmp_existing_recs (
        listing_id INT PRIMARY KEY
    );

    INSERT INTO tmp_existing_recs (listing_id)
    SELECT listing_id
    FROM tmp_recommendations;

    INSERT INTO tmp_recommendations
    SELECT
        c.car_id,
        u.listing_id,
        c.make,
        c.model,
        c.year,
        u.price,
        i.image_url,
        c.mpg
    FROM Car c
    JOIN UsedCarListing u ON u.car_id = c.car_id
    LEFT JOIN CarImage i ON i.car_id = c.car_id
    LEFT JOIN Swipe s
      ON s.user_id = p_user_id
```

```
         AND s.listing_id = u.listing_id
        LEFT JOIN tmp_existing_recs er
          ON er.listing_id = u.listing_id
        WHERE s.listing_id IS NULL
          AND er.listing_id IS NULL
        ORDER BY RAND()
        LIMIT remaining;

        DROP TEMPORARY TABLE IF EXISTS tmp_existing_recs;
    END IF;

    SELECT * FROM tmp_recommendations LIMIT p_limit;

    DROP TEMPORARY TABLE IF EXISTS tmp_recommendations;

END$$

DELIMITER ;
```

Our stored procedure was helpful because getting a recommendations list was a query that needed to be called at multiple points in our project. By making it a stored procedure, we were able to reuse this code throughout our project. It takes two parameters, userID, and num cars to return. It will return cars based on a user's likes, selecting cars of the same make and model, then sorting them first on the year (newest first) and the price (lowest first). If the engine runs out of cars to recommend, it just fills the rest of the num cars to return with random cars to ensure there are enough recommendations.

Trigger:

```
DELIMITER $$

CREATE TRIGGER swipe_update_timestamp
BEFORE UPDATE ON Swipe
FOR EACH ROW
BEGIN
IF NEW.action <> OLD.action
THEN SET NEW.created_at = NOW();
END IF;
END$$
DELIMITER ;
```

The trigger was helpful because it updated the time at which a swipe was executed right before it was updated in the database. It is important to update these to the current times so that we are storing the data correctly. While the attribute "created_at" is now semantically less useful than before, we decided it was more useful to store the time corresponding to the most recent action, rather than the first action.

Transaction level:

```
try {
  // Explicitly set the isolation level for this transaction
  await conn.query("SET TRANSACTION ISOLATION LEVEL REPEATABLE READ");
  await conn.beginTransaction();

  // 1) Log what we're removing (advanced query: aggregation + GROUP BY)
  const [logInsert] = await conn.query(
    `
    INSERT INTO UserActionLog (user_id, removed_likes, removed_passes, removed_listings)
    SELECT
        s.user_id,
        SUM(CASE WHEN s.action = 'LIKE' THEN 1 ELSE 0 END) AS removed_likes,
        SUM(CASE WHEN s.action = 'PASS' THEN 1 ELSE 0 END) AS removed_passes,
        COUNT(DISTINCT s.listing_id) AS removed_listings
    FROM Swipe s
    WHERE s.user_id = ?
    GROUP BY s.user_id
    `,
    [user_id]
  );

  // 2) Archive all this user's swipes (advanced query: multi-join INSERT...SELECT)
  const [archiveResult] = await conn.query(
    `
    INSERT INTO SwipeArchive (user_id, listing_id, action, created_at, archived_at)
    SELECT
        s.user_id,
        s.listing_id,
        s.action,
        s.created_at,
        NOW() AS archived_at
    FROM Swipe s
    JOIN UsedCarListing u ON s.listing_id = u.listing_id
    JOIN Car c ON u.car_id = c.car_id
    WHERE s.user_id = ? AND s.action = 'LIKE'
    `,
    [user_id]
  );

  // 3) Delete the original LIKE rows
  const [deleteResult] = await conn.query(
    `DELETE FROM Swipe WHERE user_id = ? AND action = 'LIKE'`,
    [user_id]
  );

  await conn.commit();

  res.json({
```

The transaction level is important because resetting likes requires multiple steps (log, archive, and delete) that must be all or nothing, as otherwise the history could be inconsistent. We set REPEATABLE READ for the transaction so that the counts we log and the rows we archive are consistent with each other, even if other swipes are submitted concurrently.

Constraints:

```
mysql> ALTER TABLE UsedCarListing
    -> ADD CONSTRAINT check_price_positive CHECK (price >= 0);
Query OK, 1835 rows affected (0.75 sec)
Records: 1835  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE Car
    -> ADD CONSTRAINT check_realistic_year CHECK (year >= 1886);
Query OK, 25727 rows affected (1.26 sec)
Records: 25727  Duplicates: 0  Warnings: 0
```

The constraints were important because they ensured that no bad data entered the system. They checked to make sure that all prices were positive (>= 0) and they also made sure that the earliest year for a car was on or after 1886. The reason 1886 was chosen was because after a bit of research we saw that this was the date of the earliest commercial vehicle.

Technical Challenges:

Ryed Badr:
I initially had difficulties setting up the project. Then another challenge was debugging CRUD operations. We all used different operating systems (Linux, Mac, and Windows) for this project, and so creating an environment that allowed for seamless workflow between all of us proved to be difficult. Things would work on some devices but not on others. Could have been recommended to use a Docker container to run the code potentially. Then, when debugging CRUD operations, I ran into some issues that took some time to figure out. I read the errors in the console as well as used print statements to work through debugging.

Brian McManus:
Our data came from a variety of sources and had to be cleaned. Some datasets had partial information for cars and others had other information we needed about those same cars, so we needed a way to automate combining these datasets. This proved to be difficult because there were no common keys to join them on. The names of cars had some partial matches but were stored differently across different datasets (some stored the full car name, others were abbreviated) as well as genuine errors with the datasets, like the brand name incorrectly being split between the brand and the make (e.g. MERCEDES,BENZ GLE 63 S instead of MERCEDES BENZ, GLE 63 S). I used python with partial name matching combined with manually identifying exceptions in order to combine these datasets and get all the required information. Getting the right balance between finding enough matches and making sure the matches were correct took significant amounts of manual effort in tuning and validating, but in the end we managed to obtain a sufficiently large collection of new and used cars with corresponding images.

Josh Esrig:
The biggest technical challenge for me was connecting to GCP on my local computer and using that to make read requests from the database and display results on the frontend. I found online tutorials and did research into how to make the connection and eventually got it working. This

was the most time consuming aspect of the project, as errors continuously arose. Overall it was a very complicated process to get everything configured correctly with the appropriate permissions, but it was rewarding to have our website and database both deployed in the cloud.

Our future plans include adding a feature to actually post a listing. Right now it is all imported data from other sources that act as the listings. In the future, we want to allow users to post their own cars on the app. Additionally, a “car meetup” feature could be cool. This would act more like tinder where you set up a profile for your car and the swipes indicate if you are open to attending a car meet together. This would add a social aspect to our app.

For our division of labor, we made sure to have regular Google Meet calls in order to discuss progress and what everyone should be working on. Further, we had a text group chat to check in on progress. Instead of splitting up the work into front-end, back-end groups, we split it up by features so that we would all get the experience of working on every section. Below is the final breakdown of the work. Everyone helped with setup and planning, however, these are the more individual contributions

Ryed Badr:
Developed the preferences filtering feature and modal.
Added the constraints and tables to the database.
Developed the intuition behind the advanced queries.
Frontend work
Major report contributions

Brian McManus:
Made the initial connection between front-end and back-end.
Major initial setup contributions (GCP)
Cleaned the data.
Implemented the keyword filtering feature
Major frontend work
Report contributions and video contributions
Like and dislike buttons
Trigger/transaction

Josh Esrig:
Developed the user creation/logging in features
Implemented the advanced queries into the filtering buttons
Added history
Major frontend work
Report contributions
Stored procedure