DiscoverEase: Tailored App Recommendations at Your Fingertips

GROUP 13 TEAM MEMBERS

- Swarn Gaba (sgaba@iu.edu)
- Gandhar Ravindra Pansare (gpansar@iu.edu)
- Chaithra Lal Nair (cnair@iu.edu)

DATABASE DESIGN

Introduction

This database design document sheds light on the meticulous process of structuring our database tables and the comprehensive normalizations applied in our pursuit of creating an app recommendation system. We are setting out to redefine the app discovery experience, making it more personalized and insightful. Our objective is to significantly enhance user interaction with the system, ensuring that recommendations are not only relevant but also resonate with the users' preferences and needs.

Conceptual Diagram/Schema

In the context of our SQL-based application recommender system, we have developed a conceptual schema to illustrate the structure of our database. This schema is presented through visuals across two stages of design: Figure 1 illustrates the Entity-Relationship (ER) diagram, detailing how entities within our database relate to one another, and Figure 2 presents the tables in their normalized form following the normalization process.

The Figure 1 below represents an Entity-Relationship (ER) diagram for an SQL-based application recommendation system. In this diagram:

- **Application Entity:** At the center is the Application entity, which is the primary entity in this schema. It has attributes such as App_Id, Size, Genre, Age_group, and Avg_user_rating. These attributes store the unique identifier for each application, its size on disk, the genre it belongs to, the target age group, and the average rating given by users.

    - **Versions Attribute:** Connected to the Application entity is the Versions relationship. This indicates that each application can have multiple versions. Attributes of this relationship include iOS_version and the App_version.

- **Date Attribute:** The Date attribute encompasses Released_date and Updated_date. These attributes capture the initial release date of the app as well as the dates on which any updates were made, offering a timeline of the app's evolution.

- **Developer Entity:** The Developer entity is connected to the Application entity by a developed_by relationship, which signifies which developer or development team has created the application.

- **Pricing Entity:** There is also a Pricing entity connected by a costing relationship, suggesting that each application may have associated pricing information. The Pricing entity has Price and Currency attributes.

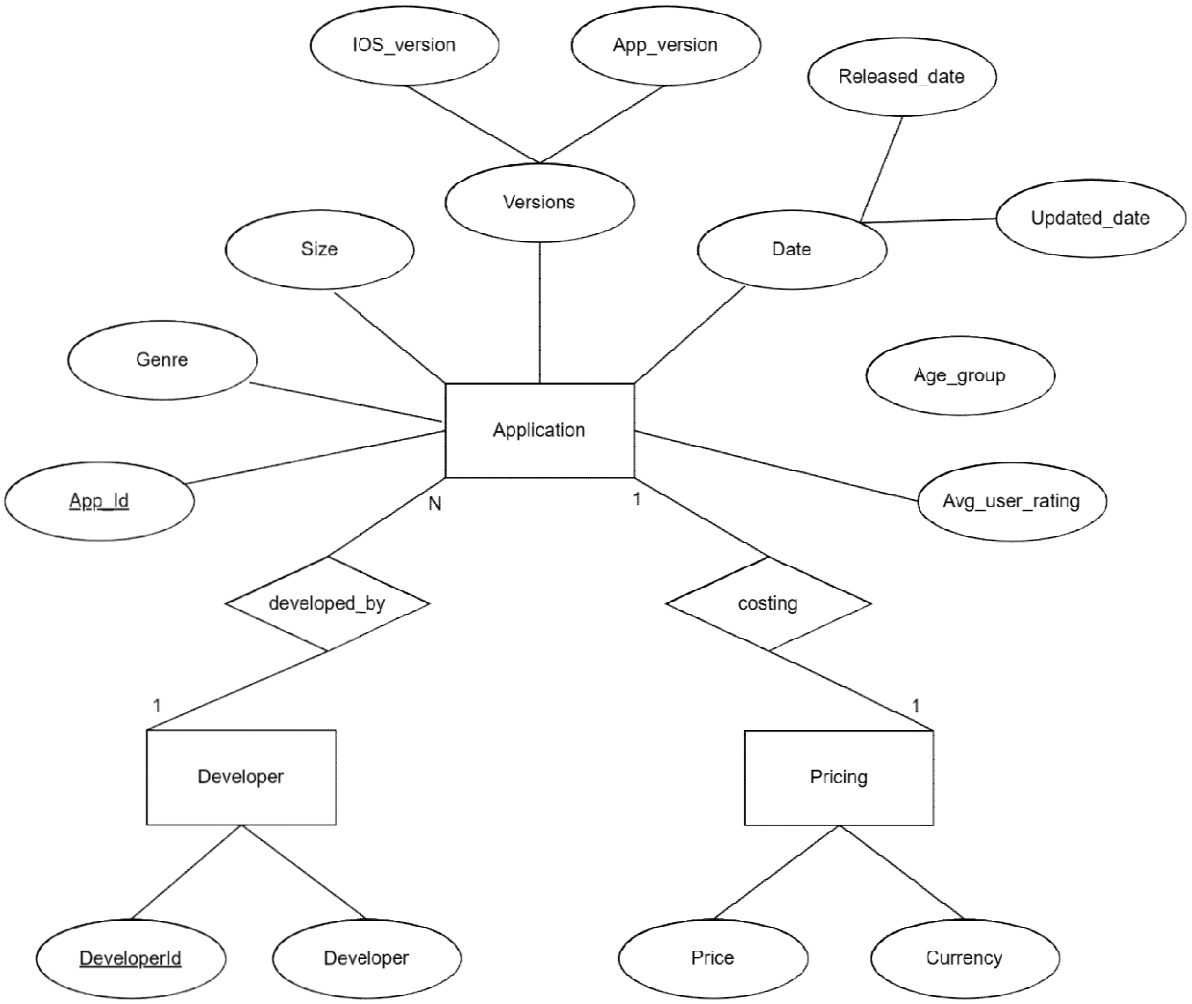

Figure: 1

The "N" near the Application entity and "1" near the Developer and Pricing entities imply the cardinality of the relationships. So, one developer can develop many applications (1-to-N), and an application could have one set of pricing information (1-to-1).

This ER diagram is a high-level conceptual representation that serves as a blueprint for the relational database structure underlying the app recommendation system we will be going to create. It shows how data is organized and interrelated, which is foundational for executing SQL queries to retrieve, insert, or manage the application data within the system.

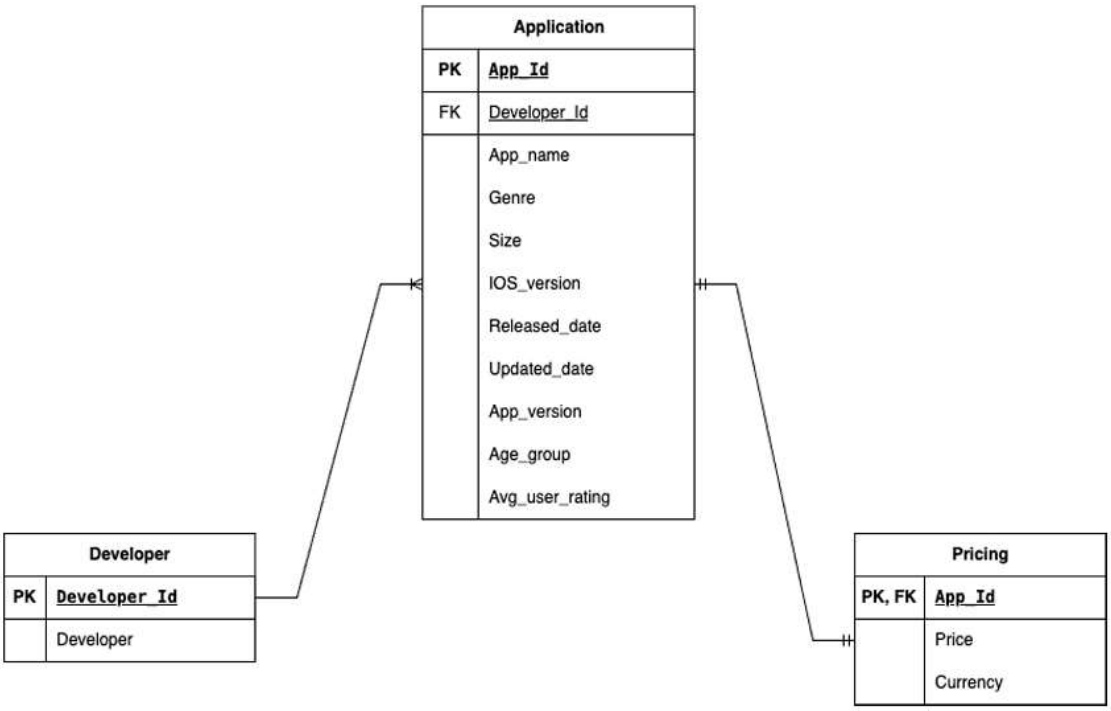

Figure: 2

Figure 2 showcases the tables after they have been normalized. This process enhances data storage efficiency and reduces duplication, leading to the creation of three separate tables: 'Applications,' 'Developers,' and 'Pricing.' Each table holds information specific to its entity, thereby streamlining and refining the database architecture for our app recommendation system.

Database Constraints

In order to preserve the consistency and integrity of the data, rules and conditions known as database constraints are applied to the data in a database. By limiting the insertion of erroneous or inconsistent data, these restrictions aid in ensuring that the data stored in the database complies with specific guidelines. Below are the various constraints implemented within our database schema to safeguard data integrity and enforce data consistency.

Primary Key Constraints: These constraints are vital as they ensure that each record within a table can be uniquely identified. This is accomplished through a specific column, or a combination of columns, known as the primary key. Not only do primary keys prevent duplicate entries, but they also facilitate the formation of relationships between tables within a relational database framework.

- App_id serves as the primary key within the Applications table.
- Developer_id is designated as the primary key within the Developers table.
- The Pricing table also uses App_id as its primary key, indicating a shared identification system with the Applications table.

Foreign Key Constraints: These constraints are the linchpins that interconnect tables within a relational database. By ensuring that a set of values in one table corresponds to values within another table (typically the primary key), foreign key constraints enforce referential integrity. This connection between tables is essential for preserving the coherent relationships across the database and preventing any data discrepancies.

- In the Applications table, Developer_Id is assigned as a foreign key, establishing a direct reference to the Developers table.
- The Pricing table identifies App_Id as its foreign key, linking back to the Applications table for reference.

Not Null Constraints: These constraints are critical for ensuring that specific columns within a table are never left empty, thus enforcing the requirement for data presence in those columns.

- Within the Applications table, columns such as App_name and Genre are mandated to be non-null.
- The Developers table enforces a non-null requirement for the Developer column.
- The Pricing table similarly dictates that Price and Currency fields must always contain data.

Code:

(Refer to the Project_Part_2 Jupyter Source / HTML File for Complete Details of Code Including Authorship)

1. Data Cleaning

An essential part of preparing data is cleaning it, which guarantees the dataset is correct, consistent, and ready for analysis. The steps involved in our data cleaning are

- **Checking for null values:** Our first step is to find and address missing values in the dataset.

Checking for NaN (Missing) Values

```
# Counting the number of NaN (Missing) values in each column of the appleAppData DataFrame
missing_values_count = appleAppData.isna().sum()

# Printing the number of NaN (Missing) values in each column
print("NaN (Missing) Values In Each Column :\n")
print(missing_values_count )
```

```
NaN (Missing) Values In Each Column :

App_Id                        0
App_Name                      1
AppStore_Url                  0
Primary_Genre                 0
Content_Rating                0
Size_Bytes                  224
Required_IOS_Version          0
Released                      3
Updated                       0
Version                       0
Price                       490
Currency                      0
Free                          0
DeveloperId                   0
Developer                     0
Developer_Url              1109
Developer_Website        643988
Average_User_Rating           0
Reviews                       0
Current_Version_Score         0
Current_Version_Reviews       0
dtype: int64
```

- **Dropping Unnecessary Columns:**
  Not all columns in the original dataset are relevant to our project's goals. This step involves assessing each column's utility and removing those that are not needed, thus simplifying the dataset and focusing on the most important information. So, we drop the columns names AppStore_Url, Free, Developer_Url, Developer_Website, Current_Version_Score, and Current_Version_Reviews.

Dropping Unnecessary Columns

```
# Dropping columns 'Developer_Website', 'Developer_Url', 'Reviews', 'Current_Version_Reviews',
# 'Current_Version_Score', and 'AppStore_Url' from the appleAppData DataFrame

appleAppData.drop(['Developer_Website', 'Developer_Url', 'Reviews', 'Current_Version_Reviews', 'Current_Version_Score', 'AppStore_Url'], axis = 1, inplace = True)
```

```
# Retrieving the column names of the appleAppData DataFrame
appleAppData.columns
```

```
Index(['App_Id', 'App_Name', 'Primary_Genre', 'Content_Rating', 'Size_Bytes',
       'Required_IOS_Version', 'Released', 'Updated', 'Version', 'Price',
       'Currency', 'Free', 'DeveloperId', 'Developer', 'Average_User_Rating'],
      dtype='object')
```

- **Genre-Based Data Sampling:**
  To ensure the dataset represents a wide variety of apps, we sample the data based on app genres. This is done mainly to balance the number of apps from each genre to avoid bias toward any particular type of app.

Genre-Based Data Sampling

```
# Conducting data sampling based on genre
# To obtain approximately 2000 rows, this method ensures proportional representation from each genre

sampled_appleAppData = appleAppData.groupby('Primary_Genre', group_keys = False).apply(lambda x: x.sample(min(len(x), 2000 // len(appleAppData['Primary_Genre'].unique()))))

# Displaying the resulting sampled DataFrame
print("Sampled DataFrame:")
print(sampled_appleAppData)
```

- **Verifying Column Data Types:**
  It's important to ensure that each column in the dataset has the appropriate data type (e.g., numerical, string, datetime). This step verifies data types and makes adjustments as necessary, facilitating accurate analysis and database storage.

```
# Displaying Information about Sampled App Store Data
sampled_appleAppData.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1976 entries, 5714 to 1222972
Data columns (total 15 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   App_Id                1976 non-null   object
 1   App_Name              1976 non-null   object
 2   Primary_Genre         1976 non-null   object
 3   Content_Rating        1976 non-null   object
 4   Size_Bytes            1976 non-null   float64
 5   Required_IOS_Version  1976 non-null   object
 6   Released              1976 non-null   object
 7   Updated               1976 non-null   object
 8   Version               1976 non-null   object
 9   Price                 1976 non-null   float64
 10  Currency              1976 non-null   object
 11  Free                  1976 non-null   bool
 12  DeveloperId           1976 non-null   int64
 13  Developer             1976 non-null   object
 14  Average_User_Rating   1976 non-null   float64
dtypes: bool(1), float64(3), int64(1), object(10)
memory usage: 233.5+ KB
```

- **Conversion:** Next, we performed the following conversion in our dataset.
    - **Transforming 'Content_Rating' Column Into Categorical Values**

```
# Defining a mapping of content ratings to age groups
age_group_mapping = {"4+": "Children", "9+": "Teen", "12+": "Teen", "17+": "Adult"}

# Mapping content ratings to age groups and creating a new 'Age_Group' column
sampled_appleAppData['Age_Group'] = sampled_appleAppData['Content_Rating'].map(age_group_mapping)
```

```
# Dropping the 'Content_Rating' column from the sampled Apple App Data DataFrame
sampled_appleAppData.drop(columns=['Content_Rating'], inplace=True)
```

    - **Converting 'Size-Bytes' Column From Bytes to Megabytes:**

```
# Convert the 'Size_Bytes' column from bytes to megabytes
sampled_appleAppData['Size_Bytes'] = sampled_appleAppData['Size_Bytes'] / (1024 * 1024)

# Round the 'Size_Bytes' column to 2 decimal places
sampled_appleAppData['Size_Bytes'] = sampled_appleAppData['Size_Bytes'].round(2)

# Rename the 'Size_Bytes' column to 'Size'
sampled_appleAppData.rename(columns={'Size_Bytes': 'Size'}, inplace=True)
```

    - **Converting Data Types for 'Released' and 'Updated' Columns:**

```
# Converting the 'Released' column to datetime format
sampled_appleAppData['Released'] = pd.to_datetime(sampled_appleAppData['Released'])

# Converting the 'Updated' column to datetime format
sampled_appleAppData['Updated'] = pd.to_datetime(sampled_appleAppData['Updated'])
```

- **Final Step:** Saving the processed and cleaned data to the csv file.

```
# Saving the final DataFrame to a CSV file named 'sampled_appleAppData.csv' without including the index
final_data.to_csv('sampled_appleAppData.csv', index=False)
```

2. Database Creation and Data Insertion:

In this segment of our documentation, we delve into the intricate process of constructing our database and populating it with data, a critical foundation for our SQL-based application recommendation system. We will employ Python and SQL code to meticulously define the schema of three interrelated tables: applications, developers, and pricing. This meticulously designed schema provides a robust foundation for organizing data related to mobile applications, their creators, and pricing details. It prioritizes the accuracy and consistent relationships of data within the database, establishing a dependable structure for storing and accessing information.

Data for the applications table is systematically imported from an external source, ensuring the creation of a comprehensive database encompassing a wide array of mobile applications. Simultaneously, a strategic approach is adopted for the developers table, where unique developer IDs are cataloged. This method is crucial for avoiding duplicate entries, thus ensuring the database remains clean and well-organized. Meanwhile, the pricing table is subject to a focused insertion process, wherein application-specific pricing details are added and directly associated with the corresponding applications.

3. View Creation: Creating Views in the Database

In this part of our guide on designing the database, we talk about adding "views" that help users see the data in different perspectives. These views make it easier to find and understand the data by showing it from various angles.

- **TopDevelopers View:** This view shows the developers who have made the most apps. It's a great way for users to see who the busiest app makers are.

- **FreeApplications View:** If users are looking for free apps, this view helps them find all of them quickly. It makes searching for no-cost options straightforward.

- **TopRatedAppsByGenre View:** This one sorts apps by their type (or genre) and based on their rating. It helps users easily find the highly-rated applications in the categories they're interested in.

Adding these views makes the database more user-friendly, giving valuable insights and making it simpler for everyone to get the information they need. We're also planning to add more views like these in the future to make the database even more helpful, providing users with new and better ways to explore the data.

Conclusion

In conclusion, the second stage of our project focused on database design and the crucial tasks of data cleaning, insertion, and creating user views. Our work on developing a conceptual schema and normalizing tables lays the foundation for effective data management, ensuring streamlined storage and easy retrieval of information. Additionally, the introduction of user views significantly boosts the database's functionality, offering tailored insights for a diverse user base.

As we conclude this phase, our team is excited about our achievements in database design and eager to move forward with the app creation stage. This journey from establishing the database foundations to enhancing it with user-centric views demonstrates our approach towards creating an innovative and accessible platform. We look forward to unveiling our efforts, which we believe will transform app discovery into a more personal and engaging experience for users.

Overall Contribution Summary

Name	Tasks	Contribution	Average Time Spent (Hrs)
Swarn Gaba	Conceptual Schema Database Code	Coordinated and ideated the design process, defined and created entity relationships, validated schema accuracy, and ensured normalization aligns with project objectives. Aided in documentation. Primary Key Implementation - Identification of primary key constraints and oversaw the enforcement of not null constraints to ensure data completeness. Spearheaded the data cleaning process i.e. checking for missing values, dropping unnecessary columns, genre-based data sampling, and verifying column data types. Created the schema for the “applications” table and managed the insertion of clean data into “applications” table. Establishing “Top Developers” view for identifying top developers based on the number of applications they've developed.	20 hours
Gandhar Ravindra Pansare	Conceptual Schema Database Code	Executed data analysis for schema development, created entity relationships diagram and contributed to normalization. Assisted in report creation. Foreign Key Integration - Analyzed data relationships and implemented foreign key constraints to maintain referential integrity between tables. Responsible for importing necessary modules, transforming data into categorical values. Created the schema for the “developers” table and managed the insertion of clean data into “developers” table. Establishing “Free Applications” view for listing free applications along with their developers.	20 hours

Name	Tasks	Contribution	Average Time Spent (Hrs)
Chaithra Lal Nair	Conceptual Schema Database Code	Researched tech stack implications for schema, refined entities and attributes, and contributed to schema normalization. Led report creation. Constraints Validation - Verified and validated all constraints across tables for consistency, ensuring the integrity and reliability of data relations. Focused on converting data types, renaming all columns and resetting index of the dataframe. Created the schema for the "pricing" table and managed the insertion of clean data into "pricing" table. Establishing "Top-Rated Apps per Genre" view for identifying the highest rated applications by genre.	20 hours