“AppFusion: Where Apps Align with Your Desires”

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. Our project revolves around revolutionizing how people discover and receive app recommendations. This approach, designed to elevate user engagement and deliver well-informed recommendations, places customization at its core.

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. The figures depict the schema in the two phases of design: Figure 1 depicts the Entity-Relationship (ER) diagram, while Figure 2 shows the normalized tables after normalization.

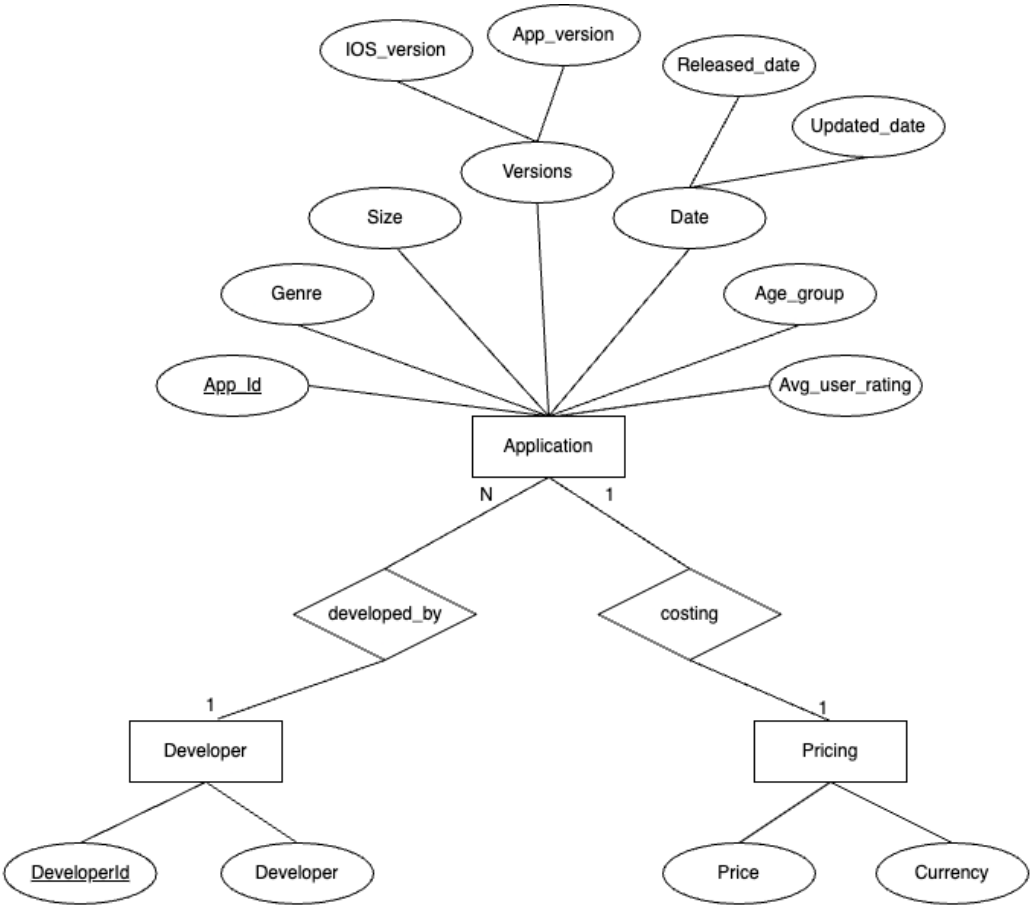

Figure 1

In Figure 1, we've identified three key entities: 'application,' 'developer,' and 'pricing.' Relationship types interconnect these entities. A many-to-one relationship between 'application' and 'developer' indicates that a single developer can be associated with multiple applications. Additionally, there's a one-to-one relationship between 'application' and 'pricing,' signifying that each application corresponds to a specific pricing structure.

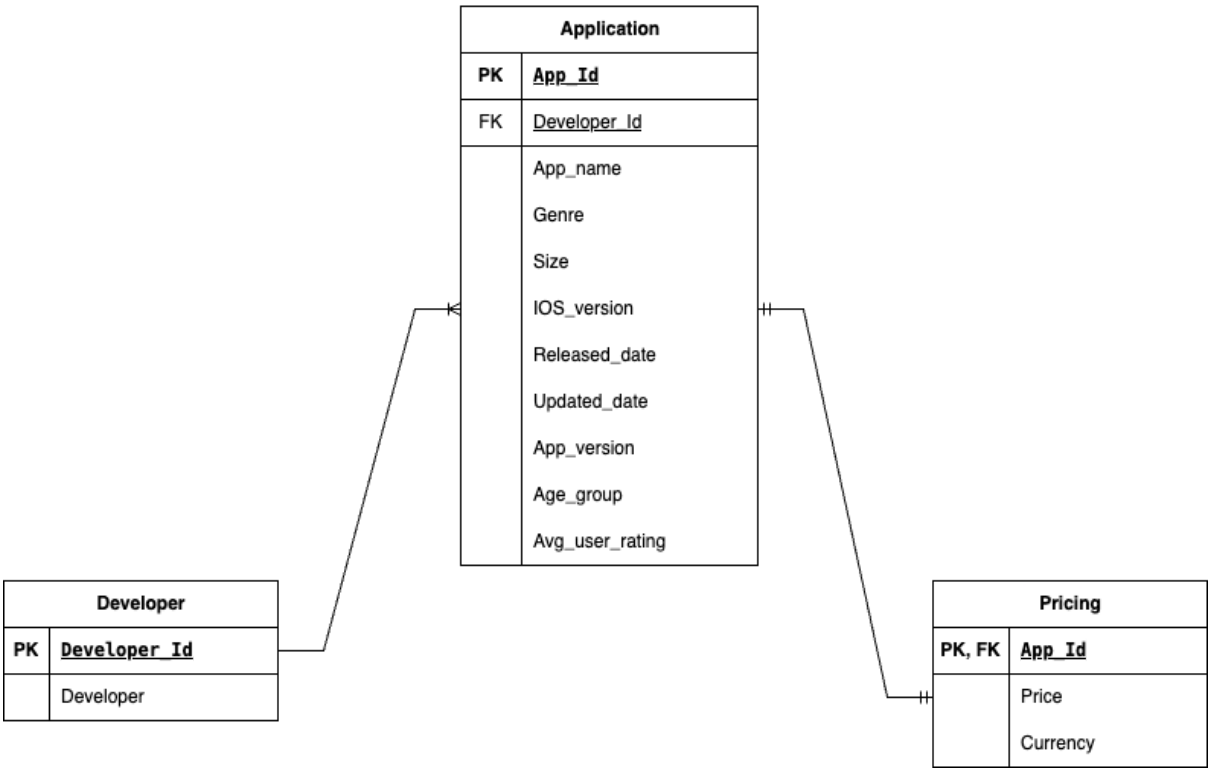

Figure 2

Figure 2 represents the normalized tables post-normalization. This step ensures efficient data storage and minimizes redundancy, resulting in three distinct tables, one for each entity: 'Applications,' 'Developers,' and 'Pricing.' These tables contain data that corresponds to their respective entities, creating a more organized and optimized database structure for our application recommender system.

Database Constraints

What are 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.

Constraints that we followed:-

- **Primary Key Constraints**
  The primary key constraint ensures that each record in a table is uniquely identified by a specific column or combination of columns. This uniqueness prevents duplicate entries and facilitates the establishment of relationships between tables in a relational database.
    - App_id is the primary key for the Applications tables.
    - Developer_id is the primary key for the Developers table
    - App_id is the primary key for the Pricing table

- **Foreign Key Constraints**
  A foreign key constraint establishes a link between tables in a relational database by ensuring that the values in a specific column (or set of columns) in one table correspond to the values in a referenced column (usually a primary key) in another table. This enforces referential integrity, helping maintain relationships and preventing inconsistencies in the data.
    - For the Applications table we have created Developer_Id as a foreign key, which will act as reference for the Developers table.
    - For the Pricing table we have created App_Id as the foreign key, which will act as reference for the Applications table

- **Not Null Constraint**
  Ensures that column does not have NULL values. It enforces the presence of Data in a particular column.
    - App_name and Genre in the applications table cannot be null.
    - Developers cannot be null in the Developers table.
    - Price and Currency cannot be null in the Pricing table.

Code

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 the NaN values

```
In [4]: # Check for NaN values in each column
        nan_values = Appstore_data.isna().sum()

        print("NaN values in each column:")
        print(nan_values)
```

```
NaN 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**: Certain columns might not contribute substantially to the analysis or might have repeated information. In our project, this involves the identification and removal of particular columns from the dataset which are AppStore_Url, Free, Developer_Url, Developer_Website, Reviews, Current_Version_Score, and Current_Version_Reviews.

Droping columns which are not needed

~ appstore_url, Developer_URL, Developer_Website, Reviews, Current_Version_Score, Current_Version_Reviews

```
In [5]: #here we are dropping the columns which are not needed
        Appstore_data.drop(['Developer_Website', 'Developer_Url', 'Reviews', 'Current_Version_Reviews', 'Current_Version_Score'
```

```
In [6]: #the columns that are in the dataset
        Appstore_data.columns
```

```
Out[6]: 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')
```

- **Performing Sampling Based on Genre**: Genre-based dataset sampling can help produce a more balanced subset of the data.

Performing sampling based on genre

```
In [7]: # If you want exactly approx 2000 rows, you can use the following approach
        sampled_appstore = Appstore_data.groupby('Primary_Genre', group_keys=False).apply(lambda x: x.sample(min(len(x), 2000 /

        # Display the sampled DataFrame
        print("Sampled DataFrame:")
        print(sampled_appstore)
```

- **Checking for Column Datatypes**: It is essential to check that the datatypes of each column are suitable for the kinds of data that they correspond to. Columns that might require conversion are identified in this step.

checking the columns datatype

```
In [12]: #info about each column of the sampled_appstore
         sampled_appstore.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1976 entries, 73935 to 989812
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.
  - Converted the Content_Rating column to a categorical value.

Convert content_Rating to categorical values

```
In [14]: # Create a mapping dictionary for Content_Rating
         rating_mapping = {"4+": "Children", "9+": "Teen", "12+": "Teen", "17+": "Adult"}

         # Create a new column 'Age_Group' based on 'Content_Rating'
         sampled_appstore['Age_Group'] = sampled_appstore['Content_Rating'].map(rating_mapping)
```

```
In [15]: #dropping the Content_Rating
         sampled_appstore.drop(columns=['Content_Rating'], inplace=True)
```

  - Convert the Size column from Bytes to Megabytes.

Convert size bytes to megabytes

```
In [19]: # Assuming 'Bytes_Column' is the name of the column containing values in bytes
         sampled_appstore['Size_Bytes'] = sampled_appstore['Size_Bytes'] / (1024 * 1024)  # Convert bytes to megabytes

         # Optionally, round the values to a specific number of decimal places
         sampled_appstore['Size_Bytes'] = sampled_appstore['Size_Bytes'].round(2)  # Round to 2 decimal places

         #renaming the columns to Size to Size_Bytes
         sampled_appstore.rename(columns={'Size_Bytes': 'Size'}, inplace=True)
```

  - Convert the columns "Updated" and "Released" to Datetime datatype.

Changing the datatype for columns Released and Update

```
In [22]: #changing the Released and Updated columns to datetime
         sampled_appstore['Released'] = pd.to_datetime(sampled_appstore['Released'])
         sampled_appstore['Updated'] = pd.to_datetime(sampled_appstore['Updated'])
```

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

```
In [29]: # Save DataFrame to CSV file
         final_df.to_csv('sampled_appstore.csv', index=False)
```

2. Database Creation & Data Insertion:

The provided Python and SQL code defines the schema for three interconnected tables: 'applications,' 'developers,' and 'pricing.' The 'applications' table captures details about mobile apps, including App_Id, App_name, Developer_Id, Genre, and other essential attributes. The 'developers' table records developer information and is linked to 'applications' through the Developer_Id. Additionally, the 'pricing' table stores app pricing data associated with the 'applications' table. This schema establishes a robust basis for storing and retrieving information about mobile applications, their developers, and pricing details, ensuring data integrity and relational consistency within the database.

This section of the database design documentation describes the data insertion process for the 'applications,' 'developers,' and 'pricing' tables. Data is systematically inserted from an external source for the' applications' table, while the 'developers' table captures unique developer IDs to avoid duplicates. In the 'pricing' table, application-specific pricing information is linked. This insertion process maintains data accuracy and keeps the database up-to-date, ensuring its reliability as a valuable resource for mobile application-related information.

3. View Creation:

This section of the database design documentation introduces user views, which provide different perspectives on the data within the database.

- The "TopDevelopers" view highlights the top developers based on the number of applications they've created, aiding users in recognizing prolific developers.
- The "FreeApplications" view presents a catalog of free applications, assisting users in identifying cost-free options.
- The "TopRatedAppsByGenre" view reveals highly-rated applications by genre, simplifying the process of finding quality apps.

These user views enhance the database's usability, offering valuable insights and making data retrieval more efficient for various stakeholders. Additionally, we plan to expand the selection of user views in the future, further enriching the database's functionality and providing users with more valuable ways to interact with the data.

Overall Contribution Summary

Name	Task	Contribution
Anushree	Conceptual Schema Database Code	Ideated schema. Designed and created ER diagram. Documentation Formulated constraints to reduce data redundancy. Contributed to data cleaning. Performed CRUD on the database and designed user views to aid in faster retrieval of data
Kaushik	Conceptual Schema Database Code	Helped in the ideation of the creation of tables. Documentation. I have preprocessed and cleaned the data in the CSV file that we are using to populate the database. The file contained a lot of null values, along with some unknown characters that could have resulted in difficulties while populating the database. To ensure smooth insertion of the data in the database, this step was necessary. Designed user view.
Shubhangi	Conceptual Schema Database Code	Discussed the schema structure Formulated data cleaning and processing steps Code for data sampling, cleaning and storing the processed data in the csv file. Designed user view.

Conclusion

In conclusion, the second stage of the AppFusion project was focused on database design and implementing key processes such as data cleaning, insertion, and user views. The conceptual schema and normalized tables ensure efficient data storage and retrieval, while the user views enrich the database's functionality. As we conclude this stage, our team is excited about our database design and ready to move to the next step for app creation. We look forward to unveiling the culmination of our efforts, providing users with an innovative and user-friendly platform for discovering personalized app recommendations.