OD Project - Android App Recommender

Aitor Ortiz de Latierro Javier Ferrando Marcel Porta Vallés

June 2019

Abstract

Graph databases are ideal for automatic processes like **recommendation systems**. In this project we describe how we would design an App recommender using them. We formally define the problem, design how we would organize data from different sources through an integrated repository, and fully explain our data analysis process.

We provide a PoC of the system at `https://github.com/Marcelpv96/OD-Final-Project`.

1 Business Idea

We want to develop a **recommender of Android applications**. The system will recommend apps to users based on their profile. For example, if someone is downloading all the time applications of category `Games`, the recommender will suggest the best rated games.

To deal with this task we will use **a couple sources of information**:

- **Amazon Reviews**[1] ⇒ Amazon is the biggest e-commerce in the world and it also has enormous amounts of data. This data source will provide to us 2.638.173 reviews with all its metadata. The information is split in two `JSON` files:
  - **Reviews** ⇒ `reviewerID`, `productID`, `reviewerName`, `reviewContent`...
  - **Metadata** ⇒ `productID`, product relationships (`AlsoBought`, `BoughtTogether`, `AlsoViewed`, `BoughtAfterViewing`), `categories`...
- **Google Play Store**[2] ⇒ `CSV` file with information of about 4.000.000 Android apps: `name`, `datePublished`, `numDownloadsMin`, `fileSize`, `packageName`, `price`, `aggregateRating`, `ratingCount`...

We have concluded that the best graph family for this kind of problem is **Property Graphs**. We will work under the *Closed-World* assumption and won't require much logical reasoning nor linking results through the *semantic web*. Moreover we will need efficient data structures and algorithms to do traversals for which this kind of databases are very well suited.

For our PoC we will use `Neo4j`, the **most popular graph database** and the one we used in the course's `Property Graphs Laboratory`. Although we won't use its graph algorithms library we would surely use in a full version of our product.

[1] `https://jmcauley.ucsd.edu/data/amazon`
[2] `https://www.kaggle.com/lava18/google-play-store-apps`

2 Data Integration

We have two different sources of information, with different formats and different schemas so **we need to integrate them** by some means.

We have decided that the most reasonable way is using a **physical integration** approach. Both our sources are fixed *data dumps* so by developing an ETL that does the most complex work beforehand we will be able to have **high performance queries**.

Integration schema

First of all we have to design an **integration schema** that will be the logical schema of both our sources and the global schema that uses all of them.

- **Sources**
  - **Amazon**: Consists of 2 `JSON` files. One of them has app reviews information and the other one has *metadata* information for each app.

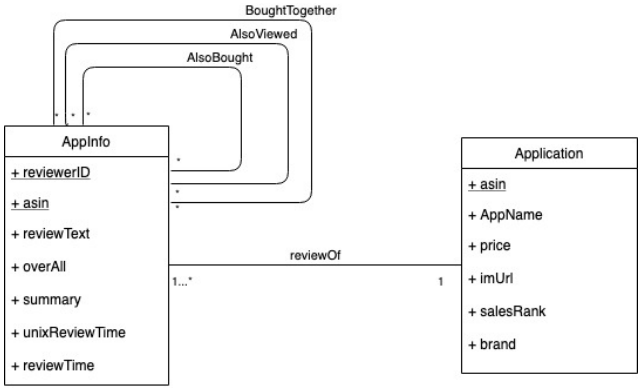

Figure 1: Amazon data source schema.

  - **Kaggle**: Consists of 1 `CSV` file. It has characteristics and statistics about all the application of the Google Play Store.

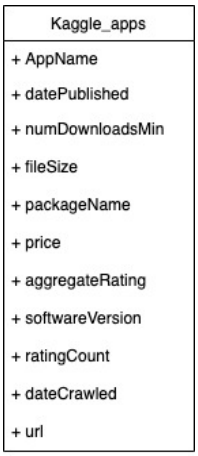

Figure 2: Kaggle data source schema.

- **Global**

  Global schema of our property graph, that combines information of both sources.

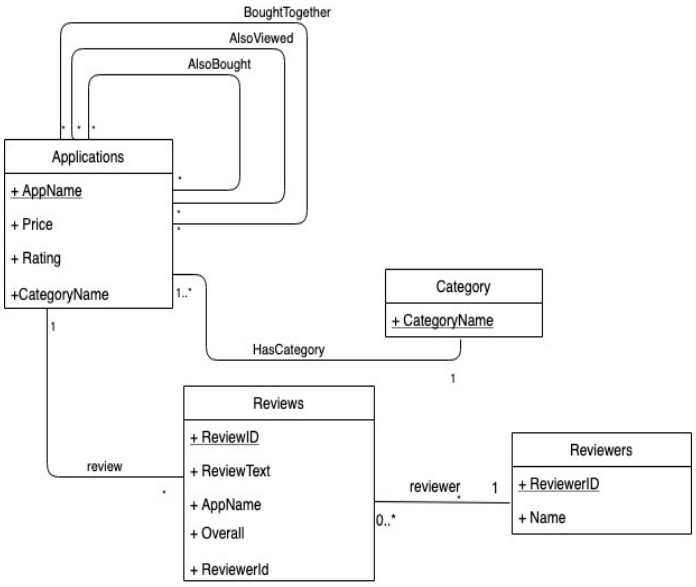

Figure 3: Global schema from integrating both sources.

ETL

We will follow the ETL process of Figure 4 to perform the data integration.

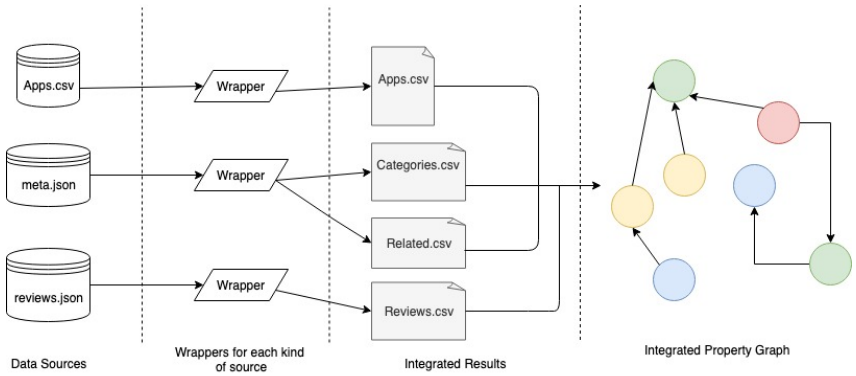

Figure 4: ETL process for both sources.

- **Extraction** ⇒ We get the data from the different sources that follow the schemas shown in Figure 1 (Amazon reviews and metadata) and 2 (Play Store information). We also do some simple pre-processing like *anomaly removal* to avoid specific loading problems in the following steps.
- **Transformation** ⇒ We generate the different `CSV` files that will form our global schema by formatting, linking and storing each table. We have found that `CSV` files are an easy and fast solution for the final loading step..

- **Loading** ⇒ We finally need to create the (property) graph database with all the relevant information. We have used the bulk load function `LOAD CSV` which has good performance for not very big datasets.

Entity Resolution

One of the most complex steps in this project has been linking the elements of both datasets through its `AppName`. The initial difficulty that we found when we wanted to code the PoC was that `Amazon` dataset **didn't actually have that field**.

Fortunately we have unique `asin` identifiers and we discovered that we could access the product information using the link `https://www.amazon.com/dp/[asin]`.
So, to work with real data, we have programmed a small *web scrapper* that gets a list of pairs `asin` / `AppName` and we've used them to link both datasets.

We have not implemented any complex entity resolution technique because, for the list of recovered `AppNames`, they were clean and easy to match (**they are not user-imputed**). However it's important to note that this may not always be true, there may be rare cases where apps change its name and they should have be taken into account in a final product.

3 Idea Exploitation

Now that we have defined our integrated data repository we can be more specific about what **kind of algorithms** we will apply to the data. We have to keep in mind that our objective is to obtain recommendations for **existing users** utilizing previous interactions.

Recommenders can be roughly divided in two different approaches:

- **Collaborative** ⇒ *"users that have common interests will probably agree in the future"* These systems recommend **items that similar users to the target have enjoyed**. They are frequently used in social media to suggest contacts/groups and e-commerce to suggest similar products to buy. They don't require a deep understanding of the product but they need some data to work well. (they suffer the *cold start* problem)
- **Content-based** ⇒ These systems are based on leveraging domain knowledge by accurately **categorizing the products with different *conceptual* labels**. Once a user interacts with the system they are categorized with the their content labels and recommendations can start by **matching users with *similar* items**. They require product understanding but give recommendations without depending on other users.

Moreover many real recommenders use **hybrid approaches** that take advantage of both.

Our proposal for this project would be combining all the available information to create the best recommender possible. We have then to use `Reviewer` preferences (through their `Overall` rating and `ReviewText` contents) and the relationships between different applications (through their `Category` and `BoughtTogether`/`AlsoViewed` connections).

Specifically, our algorithm would consist of the following steps:

(1) Find users with reviewed items in common with the target (same `Application` node)

(2) Compute their target `Similarity` (using number/percentage of common items, `Overall` ratings correlation, semantic likeness of `ReviewText`...)

(3) **Find applications** with categories/relationships in common with the target ones

(4) **Compute their target** `Adequacy` (using number/percentage of common categories, average `Overall` ratings of similar apps, `Rating`, `Price`...)

(5) **Merge** algorithmically applications from **(2)** and **(4)** & remove already owned apps

Steps **(2)**, **(4)** and **(5)** are the most complex ones and they can be done in many different ways. Our approach in a real system would be to **start using heuristics** (e.g. averaging & normalizing) but quickly try to transition to supervised ML methods using past successful/unsuccessful recommendations. **Logistic regression** would be a good first step.

We should also note that some kind validation framework will be necessary when trying to estimate recommendation success metrics. We suggest using a simple time-split test set (not used to tune the heuristics or train models) to check the system accuracy.

3.1 Proof of Concept

We have implemented a **simplified prototype** of the pipeline in `recommender.cypher`.

We get all the reviewed apps of a user and count common `categories`, `relationships` and `users` for every other app. To get the app score we add these counts to its Play Store `Rating`. Although it's a rough approximation we obtain **reasonable results** (e.g. for a user that has only reviewed `Napster` our 1st recommendation is `Shuttle+ Music Player`).