Amazon Reviews ingestion

- Takeaway.com interview technical challenge -

Mr. Luca Mircea

Delivered on 13 May 2024

Premise

- Part of the selection process for an Analytics Engineer
  - Show technical skills + data management abilities + thinking/conceptualization
- Task: design data warehouse tables + ETLs to populate them
  - Design data model (i.e. the logic of the data)
  - Implement it as code
- Resolution: GitHub repository with Dockerized code
  - www.github.com/luca-mircea/amazon-reviews-ingestion

Battle plan

In practice, there is always back-and-forth between steps & reiteration.

Nonetheless, the plan (in theory) is:

1. **Have a look at the data** to understand what it represents and how it's structured
2. **Design the data model**, i.e. the logic according to which the data will be stored in the final result
3. **Implement the code** that structures the data according to the model

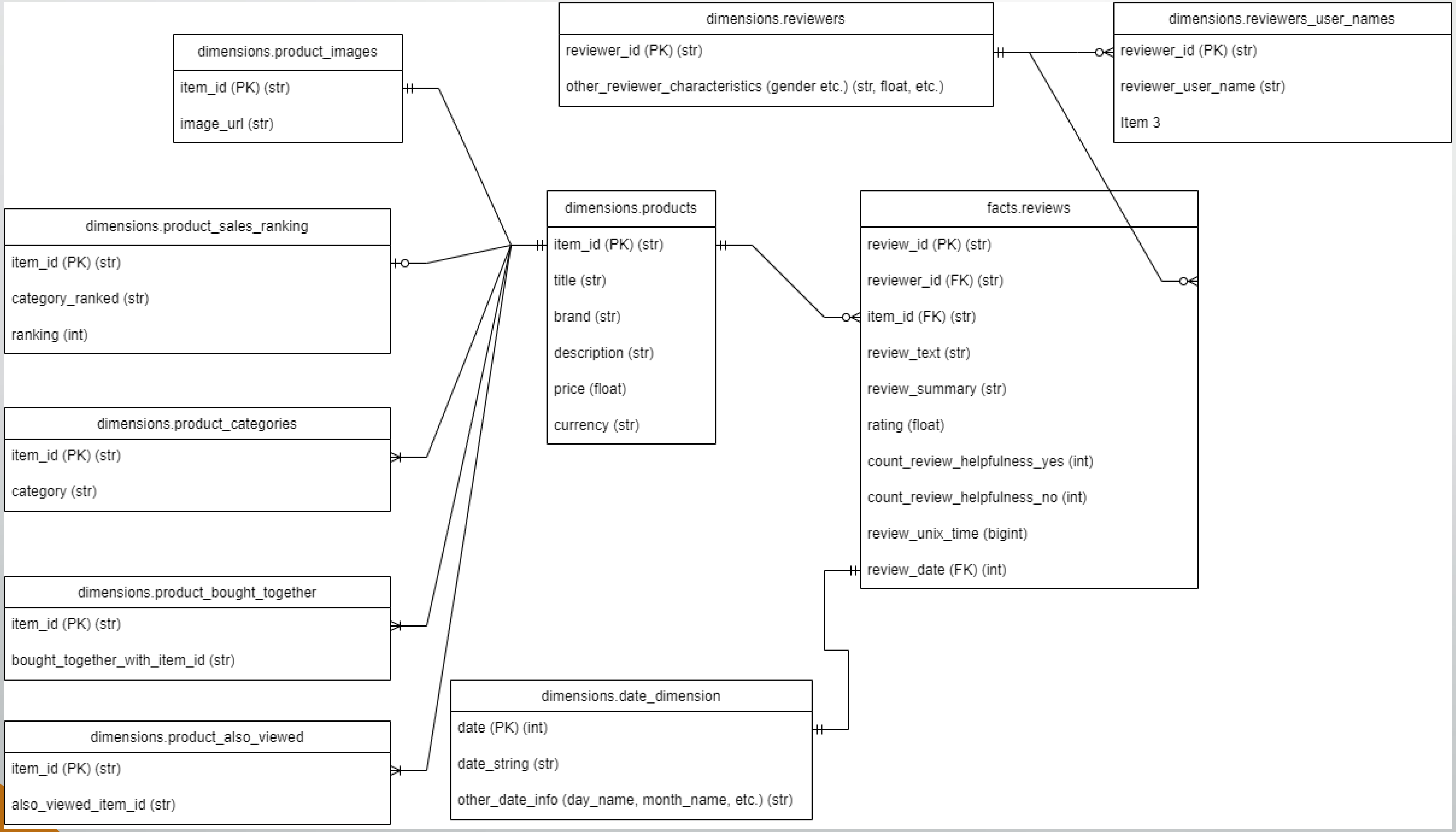
Step 1: Preliminary analysis

- Data clean to begin with:
  - Very **few NULLs**, missing in logical places
  - No duplicates
  - Mostly one per row, no messy identification process
  - Some **lack of uniformity**, e.g. sometimes missing data is NULL, other times '{ }' (in the same column)
- Main challenges:
  - **Nested** lists and dictionaries
  - **How to manage many-to-X** relationships

Step 2: Data model – logic & considerations

- The raw data consists of reviews of Amazon items + item metadata
  - The **main logical entities** underpinning the model are **products** and **reviews**
  - **Other relevant entities** that can be distilled: reviewers, date info, related_items, item_categories, rankings
- Based on the challenge requirements & present data I opted for **a hybrid model**, a combination of **Kimball and Data Vault 2.0**
  - Kimball because I'm following the **facts & dimensions** logic + **normalization**
  - Data vault 2.0 because of **hub tables** (tables with only the PK column that identifies an entity) + good way to manage edge cases and many-to-many relationships by adding many **satellite tables**:
    - **Use cases** needing the product images will rarely need the sales rank or the related items, and v-versa
    - Items belong to **multiple categories <- easier to manage** through tables different from the "main" one

Step 2 (cont.): Data model – final result



Step 3: Code implementation

- The requirements match the regular **Extract-Transform-Load (ETL)** pattern
  - Code structured in each of these steps
  - High degree of **reusability**:
    - Functions for validating data and handling nulls are **abstracted and therefore data-agnostic**, which is why they **can be copied into similar projects**
    - Using **configurations** stored in .yml files that are easy to adjust
    - **APIInteractor class** is neat for future API ingestion projects
- Code implemented to read & write data to/from multiple locations
  - Main method delivered: to and from **AWS S3 as a mock database + API endpoint**
- Code implemented with **entry points & parameters**
  - **Can be run in Airflow** to take advantage of {{ data_interval_start }} and _end macros
  - Can run every hour/day/15 min for **incremental processing**

Step 3 (cont.): Trade-offs present in code

There are always trade-offs involved when writing code:

- Most salient: **readability vs. performance**. When working with others, it's extra nice to write readable code, with the efficiency losses being minor. Hence **list comprehension instead of .map**; for-loop only as a last resort
- Testing required: list comprehension and map have different speeds depending on the data size => should **prototype & experiment** (no time now)
- With data this large, we should discuss **EMR/Spark**
- NULL handling:
  - When done right, leaving empty data saves database costs
  - There **needs to be alignment + uniformity in the company**, otherwise NULLs can create errors + confusion
  - I **preferred here to err on the side of caution and replace NULLs with clearly-wrong values** that become obvious to analysts & scientists faster

Step 3 (cont.): data processing & validation

The steps to pre-process the data were as follows:

- Reviews:
  - Created unique review_id out of item_id + reviewer_id
  - Parsed strangely formatted date string from 'MM D, YYYY' to a neat int in the format 'YYYYMMDD' (with zero-padding)
  - Parsed list of helpfulness votes into independent columns
- Metadata:
  - Removed unnamed index column
  - Flatten nested lists/dictionaries for sales_rank, categories, related items

The validations were:

- Checking for uniqueness of PKs in key tables + dropping rows where the PK is null (because useless data; there were no such instances)
  - Under normal circumstances I'd try to understand where the NULLs are coming from and see if I can fix them
- Checking if the UNIX date matches the string date
- Checking if all the rankings show up in the categories (they don't) <- this would affect the data model

Step 3 (cont.): data processing steps in ETL

Uniform steps were applied to the data (step 0. is loading it in):

1. Process the columns that need adjusting in the raw data
2. Split the data into slices that will turn into the final tables
3. Rename columns
    1. Do further processing if required, e.g. flatten nested lists or dictionaries
4. Handle NULLs (drop, fill with "Unknown" or -1, or raise errors)
5. Explicitly convert the results to specific data types
6. Upload each dataset to its corresponding table

Step 3 (cont.): if I had infinite time…

Further steps I'd take:

1. Add unit tests, especially for the final function before the upload
2. Create nicer mock DWH + API with date parameters used to slice the data into efficient-to-process chunks
3. Query the resulting data with AWS Athena or similar to check if the result matches the specification
4. (Maybe) analyze the data to see the reviews, out of curiosity

Step 3 (cont.): if I had infinite time…

Further steps I'd take:

1. Add unit tests, especially for the final function before the upload
2. Create nicer mock DWH + API with date parameters used to slice the data into efficient-to-process chunks
3. Query the resulting data with AWS Athena or similar to check if the result matches the specification
4. (Maybe) analyze the data to see the reviews, out of curiosity

Running instructions

The instructions are explained in detail in an additional document, but the gist is:

1. Clone the GitHub repo
2. Download the credentials file from the secret-sharing link, paste them into a `.txt` file called `.env`, move this to `src/credentials`
3. Build the docker image: `docker build -t takeaway-challenge .`
4. Run the desired task: `docker run takeaway-challenge python entrypoint.py --task_name process_raw_reviews_data_without_timestamps` (can replace `reviews_data` with `metadata` for the other dataset)
5. Check success with `docker run takeaway-challenge python entrypoint.py --task_name check_successful_completion`

Scheduling on AirFlow

I coded the functions & Docker to make it possible to pass arguments into the entry point. This can help with running the data on Airflow, which has really nice functionalities for implementing data processing.

- The command would be **`docker run takeaway-challenge python entrypoint.py --task_name process_raw_reviews_data_with_timestamps -- start_timestamp ‘2024-05-07 00:00:00’ --end_timestamp ‘2024-05-07 01:00:00’`**
- The command on Airflow would replace the start_timestamp with {{ data_interval_start }} and end_ with {{ data_interval_end }}

```
process_reviews_data = KubernetesPodOperator(
    **common,
    name="process_reviews_data",
    arguments=[
        "python", "src/entrypoint.py",
        "--task", "process_raw_reviews_data_with_timestamps",
        "--start_timestamp", "{{ data_interval_start }}",
        "--end_timestamp", "{{ data_interval_end }}",
    ],
    task_id="process_reviews_data"
)
```

Conclusion

The data ingestion pipeline has been built:

1. I dove into the data to understand it and get a grasp of what's in there
2. I created a data model based on what I thought made sense
3. I wrote code for the ETLs that would feed the DWH tables, optimizing for readability while staying aware of efficiency losses, and Dockerizing it for ease of use/testing and to demonstrate Airflow usage
4. Please hire me!