Big Data and Linked Data Management
Distributed Databases Coursework

Juan Casanova

February 13, 2024

Contents

1 Context

The exercise in this coursework is introduced in the context of the database and application that we have been discussing so far as an example during the course. That is, a database for managing automated distributed sales of beverages in vending machines.

In particular:

- The database has a `products` collection containing a list of products that we sell. The following is an example product document:

```
{
   "code":"coke",
   "name":"Coca-cola",
   "provider":"cocacolaco",
   "wholesale_price":0.95,
   "sale_price":1.5
}
```

- The database has a `sales` collection containing sales performed at vending machines. The following is an example sale document:

```
{
   "product":{
      "code":"pepsi",
      "name":"Pepsi",
      "provider":"pepsico",
      "wholesale_price":0.95,
      "sale_price":1.5
   },
   "vending_machine":{
      "serial_no":12345678,
      "location":"Edinburgh George Square"
   },
   "card":778582055170,
   "validation_code":4289,
   "timestamp":2023-02-14T21:54:22.788+00:00,
   "status":"validated"
}
```

- The sales are locally validated against the card data when they happen. The `validation_code` field in the sales documents indicates this (but we do not need to use it).

- There is a bulk network validation process that picks up chunks of sales that are not validated against their bank yet, from the same bank, and validates them. This changes the `status` field in the sales documents to `validated`.

- The sales collection has indices on the `card`, `timestamp`, and `status` fields, individually.

2 Goals of the coursework

2.1 Domain problem

From a domain problem point of view, what we want to do is to create another process that:

Collects product sales from the database and generates bulk orders for replacement to be sent to the providers.

In particular, we want to generate documents in a `replacement_orders` collection, such that:

- Each replacement order cannot contain products from two different providers.
- Products are listed only once for each type, with a number indicating how many are required.
- The number of products required for replacement is either exactly the same or as close to possible to the number of products sold of each type (some imprecision here is acceptable, but should be reduced as much as possible).
- All products sold are included in a replacement order for the corresponding provider.

A replacement order should look something similar to this (contains at least these fields, but may contain more if you think adding them would be useful for your approach to the problem):

```
{
  "provider": "pepsico",
  "products": [
    {
      "code": "pepsi",
      "amount": 57
    },
    {
      "code": "diet_pepsi",
      "amount": 31
    }
  ]
}
```

In order to do this, we will consider two steps to the exercise:

1. Generate a new replacement order with an adequate set of products based on sales.
2. Update an existing replacement order that has not yet been *submitted* or merge multiple replacement orderst that have not yet been *submitted*. The goal here is to send only one replacement order when we

contact the provider (which may happen weekly or monthly), and have a consolidated document in the database containing this information; even if the replacement order generation process is run multiple times before actually sending the replacement order to the provider.

2.2 Technical aspects

Even if our database does not contain a large enough amount of data to be of concern and our database infrastructure does not support full horizontal scalability, *we will work under the assumption that it does* for the purposes of this coursework. That is, your considerations should assume large amounts of products, sales, nodes on which the database is distributed, and frequency of sales. Other aspects can be assumed to be more limited in number.

We will aim to implement the solution in a manner that is informed, conscious of the reality of the situation and tries to use technical savvy to improve the performance in ways that are sensible.

This includes considering different aspects including:

- Volumes of data.
- How the sales collection is queried.
- How the replacement_orders collection is queried and updated.
- How this would relate to sharding and replication in the database (if applicable).
- How this would relate to the other operations we know are going on in the database concurrently (sales being generated and bulk card validation orders).
- How can we exploit the distributed nature of the database and of our program to enhance performance.
- The pros and cons of all the design decisions with respect to the business problem. **Justify your choices both from a performance and a business problem point of view.**

When designing your solution, you may, if you wish, add new documents to the database or new fields to existing documents (with a justification). You may not, however, remove fields from documents or change the common fields from existing documents in the database as described in this document.

3 Coursework pieces and grading rubric

So what are you supposed to do exactly?

Build an application that can be run to produce replacement orders from the sales in the database.

Some important clarifications:

- You may build the application in Python, using MongoDB's aggregation queries, or through any other means you want, but it should run autonomously.
- The application can be batch-run, designed to be run as a single process or to be deployed as multiple concurrent distributed agents. However, this decision should be adequately considered and justified.
- The application will be run alongside those of the other students, and as such it should be designed to work concurrently with these (*under the assumption they are correct*). You are not meant to compensate for the mistakes of other students, but your application should be designed in such a way that it does not interfere with the other students' applications and viceversa.
- In order to make the last point doable, you may work under the assumption that the providers you work with are not cared for by other students.
- In practice, other students may well interfere with your program. Try to debug this situation where possible, let me know if it's a big problem, but overall focus more on **the design** of the solution (with technical details) than on having a complete working solution. You should run your program to check that it works at least to a degree, but it's fine if the state of the database and/or other student's programs and/or some issues you cannot debug in your program are making it not run perfectly.
- To reinforce the last point, it is fine to not submit a perfect solution. Of course the more imperfect, the lower the grade, but this problem simply does not have a perfect solution. I am looking for you to show your ability to reason about this problem, the deeper the better.

The submission will consist of all the code required to run the application, and explanations and justifications for the design decisions. These explanations and justifications can be made in a separate document, as comments in the code, as markdown blocks if you use a Jupyter notebook, or similar approaches. However, these explanations and justifications should not be too short, as you not only need to justify what you did or why you code is that way, but rather, what's the overall principles in a distributed databases / algorithms sense that govern those choices, what other options you considered and why you decided for the one you implemented.

As part of these explanations and justifications, ask yourself the following questions when designing the application, and then **include a clear response to them** (possibly as part of a larger paragraph) in the support discussion you submit. If you do not address some of these questions, that will be regarded as a lack of relevant explanations and justifications:

- Which sales are you going to target each time the process is run? How are you going to query the database to obtain such sales?
    - How does this choice relate to the problem being solved?
    - How does this choice relate to effective horizontal scalability of the process?
    - How does this choice relate to the correctness of the results when running in a concurrent manner?
    - How does this choice relate to the isolation of the process with respect to other processes running concurrently?
    - How does this choice relate to the continuous insertion of new sales in the database?
    - How does this choice relate to the potential sharding and replication of the database?
- What is the best way to update existing replacement orders? Modify existing ones, or merge them as a separate process?
- How are you going to address the concerns with the correctness of the result in the way your process is run?
    - Do you need to make sure the same sale is not targetted more than once? If so, how are you going to do it?
    - Do you need to persist a state in any of the existing documents that is not persisted now? What level of transactionality do you need for handling that state?

- How does your choice of approach to ensure a level of correctness relate to its performance?
- Do you need to add extra fields to the documents to persist some sort of state? Is this efficient?
- How is your process going to run? Is it going to be a singular process, a series of concurrent equivalent workers, or are there different agents with different tasks running simultaneously?
- What could go wrong with the way your process runs? What are the consequences of it? Is it an acceptable risk? Why?

In terms of grading, it will be divided in the following way. There are 2 parts, each worth 25 points. The first one is to *generate a new replacement order*. The second one is to *update or merge existing replacement orders*. Note, however, that **you do not need to submit separate pieces of code**. You should submit a single piece of code that accounts for both cases: new replacement orders and existing replacement orders (possibly in a unified way).

Each of these two sections will have its grade split on the following aspects:

- The implementation produces the correct result - Up to 10 points.
- The implementation has no important performance pitfalls - Up to 8 points.
- The implementation has additional optimizations implemented and/or discussed - Up to 3 points.
- There are additional thoughts and justification for the choices made that go beyond what can be achieved within this implementation - Up to 4 points.

As a result of this, a sample submission could consist of:

- A Jupyter notebook file with the implementation of both parts of the problem (generate new replacement order, updating existing ones if they exist instead).
- A support document that explains the design decisions and performance issues. This document addressess all the questions presented above, directly or indirectly, discusses why the program acts the way it does, potential issues with this implementation and how and why we

chose to handle them. Finally, it discusses some additional optimizations and improvements that could be applied to the process but are beyond the scope of this implementation.