NoSQL Seasonal Rentals Project Report

Student: Yacine Ammi **Course:** NoSQL

---

1. Justification of NoSQL Model Choice

For this project, I chose a **Document Store model**, to be implemented with **MongoDB**. After analyzing the dataset and project requirements, this model was the most suitable for the following reasons:

- **Natural Data Structure:** The `listings.csv` data is inherently document-centric. Each listing is a self-contained entity with various attributes. This structure maps perfectly to a JSON-like document in MongoDB, making the data model intuitive and easy to work with.
- **Query Flexibility:** The analysis questions require filtering and aggregating on different fields within a listing (e.g., `property_type`, `number_of_reviews`). MongoDB's rich query language and aggregation framework are designed for these types of queries, making it far superior to a simple Key-Value store for this task.
- **Schema Design:** The data contains a one-to-many relationship (one host has many listings). A document model allows for **embedding** host information directly within each listing document. This denormalization is highly effective for read-heavy applications, as it eliminates the need for expensive `JOIN` operations.

---

2. NoSQL Schema Design

My proposed schema for the `listings` collection in MongoDB uses an **embedded document pattern**. I decided to embed host-related information as a sub-document within each listing.

Key Design Decisions:

- **Embedding:** The `host` object is nested inside the main listing document. This improves read performance, as all information for a listing can be retrieved in a single database call.
- **Data Redundancy:** I accepted the trade-off of storing host information multiple times. For an analytical workload where data is written once and read many times, this is a standard and effective practice.
- **GeoJSON for Location:** The latitude and longitude were structured into a GeoJSON `Point` object to enable efficient geospatial queries in MongoDB.

Example Document Schema:

```
{
  "_id": 80260,
  "listing_url": "https://...",
  "name": "Nice studio in Jourdain's village",
  "property_type": "Entire rental unit",
  "price": 150.00,
  "amenities": ["Wifi", "Kitchen", "Heating"],
  "host": {
    "id": 333548,
    "name": "Charlotte",
    "since": "2011-01-03T00:00:00Z",
    "is_superhost": false
  },
  "location": {
    "neighbourhood_cleansed": "Ménilmontant",
    "coordinates": {
      "type": "Point",
      "coordinates": [ 2.38848, 48.87131 ]
    }
  },
  "reviews": {
    "count": 206,
    "score_rating": 4.63
  }
}
```

---

3. Data Cleaning and Validation

A Python script using the `pandas` library was developed to perform data cleaning. The raw `listings.csv` file contained several quality issues that were addressed:

- **Type Conversion:** The `price` column was converted from a string (e.g., `$250.00`) to a numeric float. Boolean columns with `'t'` and `'f'` values were converted to `True` / `False`. Date columns were converted to proper datetime objects.
- **Handling Missing Values:** Missing review scores were filled with `0`. Missing `bedrooms` and `beds` were defaulted to `1`. Text fields like `description` were filled with an empty string.
- **Complex Parsing:** The `bathrooms_text` field was parsed using regular expressions to extract a numeric value. The `amenities` column, which was stored as

a stringified list, was parsed into a proper list object.

The output of this process was a `cleaned_listings.csv` file, which served as the source for the migration step.

---

4. Data Migration: Process and Challenges

The migration was planned using a Python script with the `pymongo` library to connect to the MongoDB instance running in Docker.

The initial approach was to read the cleaned CSV into a pandas DataFrame and use `insert_many()` for an efficient bulk insert. However, this process encountered significant data quality issues that prevented a complete migration:

1. **Initial Errors:** The script first failed due to common issues like `NaT` (Not a Time) and `NaN` (Not a Number) values from pandas, which are not directly supported by MongoDB's BSON format.
2. **Encoding Errors:** Further attempts revealed `UnicodeEncodeError`, indicating invalid characters in text fields scraped from the web.
3. **Persistent `InvalidDocument` Error:** After fixing the above issues, a persistent `bson.errors.InvalidDocument` error remained. This suggests that despite the cleaning steps, there were deeply embedded data corruption or incompatible data type issues (e.g., unusual numpy types) that were difficult to isolate in the large dataset.

To overcome this, I re-engineered the script for a **resilient, row-by-row insertion** inside a `try...except` block. This strategy was designed to skip individual problematic records while successfully migrating the clean ones. Unfortunately, even this robust approach failed to complete, indicating the data quality issues were more severe than anticipated.

---

5. Docker Configuration

The project environment was successfully containerized using a `docker-compose.yml` file. This configuration defined and linked two main services:

- **`mongo`**: The official MongoDB database instance. Port `27017` was mapped for local access.
- **`mongo-express`**: A web-based GUI for easy database administration and viewing, accessible on port `8081`.

Credentials were managed securely using a `.env` file, and a Docker volume was configured for the `mongo` service to ensure data persistence across container restarts.

---

6. Analysis Queries (Theoretical)

Due to the migration challenges, it was not possible to execute the queries against the database. However, this section provides the `pymongo` code that I developed to answer the analysis questions, demonstrating my understanding of MongoDB querying.

Query 1: How many listings are there for each type of property?

```
pipeline = [
    {
        "$group": {
            "_id": "$property_type",
            "count": { "$sum": 1 }
        }
    },
    { "$sort": { "count": -1 } }
]
results = list(db.listings.aggregate(pipeline))
```

Query 3: What are the 5 listings with the highest number of reviews?

```
results = db.listings.find(
    {}, # No filter
    {"name": 1, "reviews.count": 1, "_id": 0} # Projection
).sort("reviews.count", -1).limit(5)
```

---

7. Conclusion & Key Learnings

This project was a valuable practical exercise in designing and implementing a NoSQL data pipeline. While the final data migration was unsuccessful, the preceding steps—model selection, schema design, data cleaning, and environment containerization—were all completed successfully.

The primary challenge was the unexpectedly poor quality of the source data. The key learning from this project is the critical importance of robust, fine-grained data validation and error handling. In a real-world scenario, a next step would be to implement a more intensive cleaning script to isolate and either fix or discard the specific rows causing the BSON encoding errors before attempting migration again.