# Database Systems Comparison

## Project Overview

### Target

This project explores the Yelp dataset by setting up and analyzing data using both relational (PostgreSQL) and non-relational (MongoDB) databases. The goal is to gain insights into consumer experiences, business trends, and database performance comparisons.

### Results

Designed and developed two database structures in SQL and NoSQL. One a structured schema in PostgreSQL, enforcing relationships and constraints; the other leveraging MongoDB's flexibility for semi-structured data. The dataset includes businesses, users, reviews, check-ins, and tips, allowing us to perform analytical queries on business popularity, top-rated businesses, and active users.

The project involved importing and transforming JSON data, schema design, and query optimization to compare SQL vs. NoSQL efficiency. We used DBeaver for database management and PyMongo for MongoDB queries. Ultimately, this project provided hands-on experience in database setup, data manipulation, and performance analysis.

## Database Structure

### Relational Database Structure

For our relational database, we choose Postgres as our database system. We import the data from the Yelp dataset into the database following the schema described below.

#### Overview

Our database consists of 5 entity sets, each has a number of attributes, and 5 relationships connecting the entity sets. Below is a detailed description of the database:

**Entity sets and attributes**

- `Business`: Contains business data including location data, attributes, and categories;
- `User`: User data including the user's friend mapping and all the metadata associated with the user;
- `Review`: Contains full review text data including the user_id that wrote the review and the business_id the review is written for;

- `Checkin`: Checkins on a business, a way to keep track of businesses that are visited by users;
- `Tip`: Tips written by a user on a business. Tips are shorter than reviews and tend to convey quick suggestions.

**Relationships**:

- Each review is commented by exactly one user on exactly one business;
- Each tip is left by exactly one user on exactly one business;
- Each checkin is received by at most one business;
- Each business can have however many numbers of reviews, tips, and checkins.

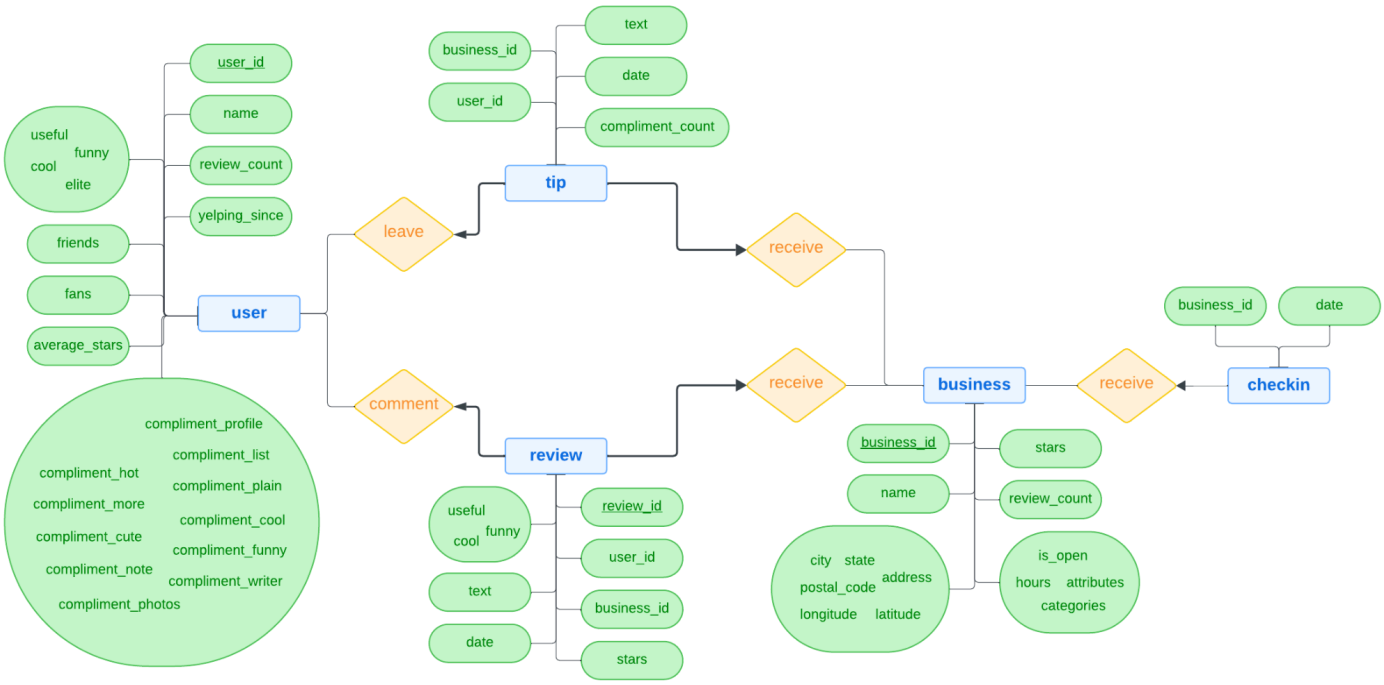
In addition to the domain-specific relationships, we also add in constraints so that data does not violate referential integrity. Eg., a review is written by a user with user_id not present in the `user` table:

**Constraints**:

- Each business has its own unique business_id stored in `business`;
- Each user has their own unique user_id stored in `user`;
- Each review has its own unique review_id stored in `review`;
- Each review is left by a user present in `user`;
- Each tip is left by a user present in `user`, to a business present in `business`;
- Each checkin is to a business present in `business`.

## ER Diagram

See below an ER diagram that draws out the database relationships, where blue rectangles are entity sets with green oval as their respective attributes (primary key attributes are underlined), yellow diamond represents an action between entity sets, and edges specify what relation (edge: no constraint, arrow: at most one, bolded arrow: exactly one).




## Database Schema

The following shows the complete database schema set up in Postgres, with all the constraints described above.

**Business:**

```
                    Table "public.business"
    Column    |         Type          | Collation | Nullable | Default
--------------+-----------------------+-----------+----------+---------
 business_id  | character varying(22) |           | not null |
 name         | text                  |           |          |
 address      | text                  |           |          |
 city         | text                  |           |          |
 state        | text                  |           |          |
 postal_code  | text                  |           |          |
 latitude     | double precision      |           |          |
 longitude    | double precision      |           |          |
 stars        | double precision      |           |          |
 review_count | integer               |           |          |
 is_open      | integer               |           |          |
 attributes   | text                  |           |          |
 categories   | text                  |           |          |
 hours        | text                  |           |          |
Indexes:
```

```
    "business_pkey" PRIMARY KEY, btree (business_id)
Referenced by:
    TABLE "checkin" CONSTRAINT "checkin_business_id_fkey" FOREIGN KEY
(business_id) REFERENCES business(business_id) ON UPDATE CASCADE ON DELETE
CASCADE
```

**User:**

```
                          Table "public.user"
       Column       |          Type          | Collation | Nullable |
Default
--------------------+------------------------+-----------+----------+------
---
 user_id            | character varying(22)  |           | not null |
 name               | character varying(100) |           |          |
 review_count       | integer                |           |          |
 yelping_since      | character varying(255) |           |          |
 useful             | integer                |           |          |
 funny              | integer                |           |          |
 cool               | integer                |           |          |
 elite              | character varying(255) |           |          |
 friends            | text                   |           |          |
 fans               | integer                |           |          |
 average_stars      | double precision       |           |          |
 compliment_hot     | integer                |           |          |
 compliment_more    | integer                |           |          |
 compliment_profile | integer                |           |          |
 compliment_cute    | integer                |           |          |
 compliment_list    | integer                |           |          |
 compliment_note    | integer                |           |          |
 compliment_plain   | integer                |           |          |
 compliment_cool    | integer                |           |          |
 compliment_funny   | integer                |           |          |
 compliment_writer  | integer                |           |          |
 compliment_photos  | integer                |           |          |
Indexes:
    "user_pkey" PRIMARY KEY, btree (user_id)
```

**Review:**

```
                    Table "public.review"
   Column    |       Type       | Collation | Nullable | Default
-------------+------------------+-----------+----------+---------
```

```
 review_id   | text             |           | not null |
 user_id     | text             |           |          |
 business_id | text             |           |          |
 stars       | double precision |           |          |
 useful      | integer          |           |          |
 funny       | integer          |           |          |
 cool        | integer          |           |          |
 text        | text             |           |          |
 date        | text             |           |          |
Indexes:
    "review_pkey" PRIMARY KEY, btree (review_id)
```

**Checkin:**

```
                        Table "public.checkin"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+--------
-
 business_id | character varying(22)       |           |          |
 date        | timestamp without time zone |           |          |
Foreign-key constraints:
    "checkin_business_id_fkey" FOREIGN KEY (business_id) REFERENCES
business(business_id) ON UPDATE CASCADE ON DELETE CASCADE
```

**Tip:**

```
                            Table "public.tip"
      Column      |            Type             | Collation | Nullable |
Default
------------------+-----------------------------+-----------+----------+---
------
 business_id      | character varying(22)       |           |          |
 user_id          | character varying(22)       |           |          |
 text             | text                        |           |          |
 date             | timestamp without time zone |           |          |
 compliment_count | integer                     |           |          |
```

## Non-relational Database Structure

We set up our non-relational database in MongoDB as it is one of the more supported NoSQL systems. Although non-relational systems do not enforce strict schemas or constraints like

relational systems do, there are ways to manage schema enforcement and relationships using tools and techniques, covered in database setup in later sections.

Similar to the database structure in Postgres, our database consists of 5 collections of respective documents. For the overview information of each collection, refer to the relational database structure section. Below is a detailed description of the collections and the types of the documents stored inside:

**Collections**

- `Business`: Stores all business documents. Each business is uniquely identified by the default `_id` generated by MongoDB and a unique `business_id` provided by the dataset. Each document stores the business's related geo-information, rating information, operating hours, categories, and additional information that varies by businesses. Each registered business in Yelp is stored as one document;
- `User`: Stores all user documents. Similar to `business`, each user is identified by `_id` and `user_id`. The document also contains the user's personal information, connection with other users, and review-related information. Each registered user on Yelp is stored as one document;
- `Review`: Stores all review documents. Similar to `business` and `user`, each review is identified by `_id` and `review_id`. The document also contains the review's text, date, and related connection to user and business. Each posted review on Yelp is stored as one document;
- `Checkin`: Stores checkins on a business per document;
- `Tip`: Stores tips written by a user on a business per document.

# System and Database Setup

## Relational Database Setup

We can summarize the setup of the relational database in three steps: system setup, data conversion, database setup.

### System Setup

We first start with system setup. We're using Postgres14 with client DBeaver and CLI. Though it is achievable to connect to Postgres without an additional client, we choose to work in DBeaver as it provides a user-friendly graphical interface that simplifies database management tasks.

We then set up a database connection on local host and a connecting user by following the setup in this github link. Afterwards, we create an empty database, `yelp`, on local host. For this project, each person set up their own database on their local connection for simplicity.

## Data Conversion

Now that we have a running database, because Postgres cannot take in JSON files as import data, we implemented a simple JSON reader that transforms the data into CSV format, conserving the original formatting and datasize.

```
# Reads in JSON per line and write into CSV
def json_to_csv(json_file, csv_file):
      column_names = get_column_names_from_json(json_file) # Get superset
of column names in json_file
      with open(json_file) as input, write(csv_file) as output:
            for line in input:
                  record = json.reads(line) # Reads json_file per line
                  output.write(record, column_names) # Write record into
CSV with corresponding column_names
```

## Database Setup

After converting all dataset into CSV files, we create entities and import data into Postgres. To ensure data type integrity, we impose data type constraints into the table schema; and to ensure referential integrity, we add in primary key constraints and foreign key constraints.

```
yelp=#
-- An example with primary keys and foreign key references
DROP TABLE IF EXISTS <table name> CASCADE;
CREATE TABLE <table name> (
      review_id VARCHAR(22) PRIMARY KEY,
      user_id VARCHAR(22) REFERENCES user(user_id) ON DELETE CASCADE ON
UPDATE CASCADE,
      business_id VARCHAR(22) REFERENCES business(business_id) ON DELETE
CASCADE ON UPDATE CASCADE,
      stars FLOAT,
      text TEXT,
      date TIMESTAMP
);
COPY <table name> (<column names>)
FROM '<CSV data file path>'
DELIMITER ',' CSV HEADER;
```

This setup stores the database locally and runs queries on the local computing system with the help of DBeaver client as a user-interface. The local setup provides easy access and a quick development/testing environment.

# Non-relational Database Setup

Since non-relational database systems support data import using JSON files, we can summarize the setup of the non-relational database setup in two steps: system setup and database setup.

## System Setup

For this section, we chose to connect MongoDB with VSCode to perform query analysis. Since our datasets contain many gigabytes, we decided to install the MongoDB Community Edition locally on our computers following the MongoDB docs. After installation, using the terminal, we created a database called `yelp`, we created a collection for each dataset that we are using, and we imported the corresponding json file.

After successfully creating the database with data loaded in, we downloaded the MongoDB extension from VSCode, connected our local host with a connection string, built a playground and started running queries.

In MongoDB, foreign key constraints and primary key constraints are not enforced like they are in SQL databases because it's a NoSQL database, thus we did not include them when importing the data into the database.

Here is a detailed code snippet that we ran in terminal to load the data into the database:

1. Start the service with `brew services start mongodb-community@8.0`
2. Run `mongosh`
3. Run `use yelp` to create new database
4. Run `exit` and run `mongoimport --db yelp --collection <collection name> --<JSON file path>`
5. Do the same for the other tables, ensure to change to a different collection name.

This sets up a `yelp` database and collections imported. The database structure is similar to below, with a default schema, which only specifies data type detected, defined by MongoDB:

```
- yelp
    - business
    - user
    - review
    - checkin
    - tip
```

## Database Setup

We want to ensure the imported data follows schema constraints similar to that of the database setup in a relational system. Though MondoDB does not support direct constraint, we can use additional system setups or enforce constraints in queries. There are two relationships we would enforce, primary keys and foreign keys:

**Primary key constraints**:

For the uniqueness constraint of a primary key, we can add indexes on collections to enforce uniqueness on the target field,

```
// First check if the target index exists
if (db.<collection>.getIndexes() == <target index name>) {
  db.<collection>.dropIndex(<target index name>);
  }
// Then create target index
db.business.createIndex({ <target index name>: 1 }, { unique: true }); //->
enforce uniqueness
```

And for the non-null constraint of a primary key, though it is possible to use a `validator` to define specific data type and rules for the field, it is costly and not scalable for large datasets. Therefore, we can add additional validation steps in task selections to make sure the output keys are valid.

**Foreign key constraints**:

Similar to the shortcomings of the data type integrity of primary keys, though it is possible, it is not scalable for MongoDB to enforce foreign key constraints on collections. Hence, additional validation steps during queries are recommended to ensure referential integrity is satisfied.

# Task Selection

## Relational Database

For the purpose of ensuring the database is correctly set up, we included three queries that tested the functionality of the `yelp` database.

### Task 1: Popular business categories1

This is an example of finding the top 10 most popular categories of business from the dataset by splitting the categories string into individual entries, removing any remaining whitespace, collecting distinct cleaned categories for each business and combining them into an array, which can be a useful query for decision making.

This also helps us understand `business` because it demonstrates information about the restaurant categories and the total number of reviews that they have for yelp, which can be useful for decision making problems.

```
SELECT
    normalized_category AS category,
    SUM(review_count) AS total_reviews
FROM (
    SELECT
```

```
        ARRAY_TO_STRING(ARRAY_AGG(DISTINCT TRIM(c)), ', ') AS
normalized_category,
        b.review_count
    FROM business AS b, regexp_split_to_table(b.categories, ',') AS c
    GROUP BY b.business_id, b.review_count
    ) AS normalized
GROUP BY normalized_category
ORDER BY total_reviews DESC
LIMIT 10;
```

| category | total_reviews |
|---|---|
| Mexican, Restaurants | 106039 |
| Event Planning & Services, Hotels, Hotels & Travel | 65877 |
| Pizza, Restaurants | 61223 |
| Italian, Restaurants | 51708 |
| Beauty & Spas, Nail Salons | 50584 |
| Chinese, Restaurants | 46056 |
| Japanese, Restaurants, Sushi Bars | 38994 |
| Italian, Pizza, Restaurants | 37856 |
| American (New), Restaurants | 35565 |
| Coffee & Tea, Food | 34701 |

Using `EXPLAIN ANALYZE`, we see that this query uses nested loop join, subquery scan, index scan and function scan, with sorting methods like top-N heapsort and external merge. We can see that the group keys are `business_id` and `normalized_category`, thus one way to improve the efficiency is to add indexes on `business_id` and `normalized_category` to optimize joins and grouping.

As an example, we splitted the big query into smaller pieces, created a temporary table, and added indexes on `business_id` and `normalized_category`, and the overall execution time decreased.

```
%%sql
DROP INDEX IF EXISTS business_id_idx;
CREATE INDEX business_id_idx ON business(business_id);

EXPLAIN ANALYZE(
SELECT
    ARRAY_TO_STRING(ARRAY_AGG(DISTINCT TRIM(c)), ', ') AS
normalized_category,
    b.review_count,
    b.business_id
FROM business AS b, regexp_split_to_table(b.categories, ',') AS c
GROUP BY b.business_id, b.review_count
```

```
);


DROP INDEX IF EXISTS normalized_category_idx;
CREATE INDEX normalized_category_idx ON
normalized_temp(normalized_category);

EXPLAIN ANALYZE (
SELECT
    normalized_category AS category,
    SUM(review_count) AS total_reviews
FROM normalized_temp
GROUP BY normalized_category
ORDER BY total_reviews DESC
LIMIT 10
);
```

## Task 2: Top rated businesses

This is a similar example that returns the business with the highest rating sorted by the number of review counts, which tells us what types of business is the most popular with a high rating.

This also helps us understand `business` because it shows information about the business names, their categories that they belong to, and their corresponding ratings and number of reviews received.

```
SELECT
    b.name AS business_name,
    ARRAY_TO_STRING(ARRAY_AGG(DISTINCT TRIM(c)), ', ') AS
normalized_category,
    b.stars AS highest_rating,
    b.review_count
FROM business AS b, regexp_split_to_table(b.categories, ',') AS c
GROUP BY  b.business_id
ORDER BY b.stars DESC, b.review_count DESC
LIMIT 10;
```

| business_name | normalized_category | highest_rating | review_count |
|---|---|---|---|
| Blues City Deli | American (Traditional), Bars, Delis, Nightlife, Pubs, Restaurants, Sandwiches | 5.0 | 991 |
| Carlillos Cocina | Bars, Breakfast & Brunch, Mexican, Nightlife, Restaurants | 5.0 | 799 |
| Free Tours By Foot | Hotels & Travel, Tours, Walking Tours | 5.0 | 769 |
| Tumerico | Gluten-Free, Mexican, Restaurants, Vegan, Vegetarian | 5.0 | 705 |
| Yats | Cajun/Creole, Caterers, Comfort Food, Event Planning & Services, Restaurants | 5.0 | 623 |
| Nelson's Green Brier Distillery | Distilleries, Food, Historical Tours, Hotels & Travel, Tours | 5.0 | 545 |
| Smiling With Hope Pizza | Italian, Pizza, Restaurants, Salad | 5.0 | 526 |
| Barracuda Deli Cafe St. Pete Beach | Breakfast & Brunch, Caribbean, Cuban, Latin American, Restaurants, Sandwiches | 5.0 | 521 |
| SUGARED + BRONZED | Beauty & Spas, Cosmetics & Beauty Supply, Day Spas, Hair Removal, Shopping, Spray Tanning, Sugaring, Tanning, Waxing | 5.0 | 513 |
| Cafe Soleil | Bakeries, Breakfast & Brunch, Cafes, Coffee & Tea, Food, French, Restaurants, Sandwiches | 5.0 | 468 |

Similarly, using `EXPLAIN ANALYZE`, we see that this query also uses nested loop join, index scan and function scan, with sorting methods like top-N heapsort again.

As we add an index on `business_id`, the execution time also decreased by 50 ms in this case.

```
DROP INDEX IF EXISTS business_id_idx;
CREATE INDEX business_id_idx ON business(business_id);

EXPLAIN ANALYZE (
SELECT
    b.name AS business_name,
    ARRAY_TO_STRING(ARRAY_AGG(DISTINCT TRIM(c)), ', ') AS
normalized_category,
    b.stars AS highest_rating,
    b.review_count
FROM business AS b, regexp_split_to_table(b.categories, ',') AS c
GROUP BY  b.business_id
ORDER BY b.stars DESC, b.review_count DESC
LIMIT 10
);
```

## Task 3: High rating active users

This last example shows the top 10 most active users who write reviews for businesses with high ratings, which indicates that they might be some sort of influencers.

This helps us understand `user` because it shows that this dataset contains the list of user names, the number of reviews they gave to businesses, and the average rating of the businesses they reviewed. This also links with the `business` table above because it can sort of tell us whether some businesses' ratings are intentionally high or low due to a certain group of users.

```
SELECT u.name AS user_name,
```

```
    COUNT(r.review_id) AS total_reviews,
    AVG(b.stars) AS avg_business_rating
FROM users u
INNER JOIN review r ON u.user_id = r.user_id
INNER JOIN business b ON r.business_id = b.business_id
WHERE b.stars >= 4.5
GROUP BY u.user_id, u.name
ORDER BY total_reviews DESC
LIMIT 10;
```

| user_name | total_reviews | avg_business_rating |
|---|---|---|
| Ken | 678 | 4.584808259587021 |
| Brittany | 569 | 4.6230228471001755 |
| Brett | 557 | 4.581687612208259 |
| Jen | 547 | 4.593235831809872 |
| Karen | 489 | 4.569529652351738 |
| Boon | 489 | 4.5531697341513295 |
| Marielle | 479 | 4.566805845511483 |
| Michael | 458 | 4.615720524017467 |
| Steven | 438 | 4.559360730593608 |
| Niki | 420 | 4.621428571428571 |

With `EXPLAIN ANALYZE`, we see that this last query is taking a long time to execute as the datasets contain many gigabytes, and so even if it's using parallel hash joins and parallel sequential scans, it's still quite difficult to join millions of rows together.

One way to improve this query is to create indexes as usual, filter out unnecessary rows with WHERE before the join, and break the big query into small queries, and this decreased the execution time by over 3000 ms.

```
DROP INDEX IF EXISTS user_id_idx;
CREATE INDEX user_id_idx ON users(user_id);
DROP INDEX IF EXISTS review_user_id_idx;
CREATE INDEX review_user_id_idx ON review(user_id);
DROP INDEX IF EXISTS review_business_id_idx;
CREATE INDEX review_business_id_idx ON review(business_id);

EXPLAIN ANALYZE(
WITH filtered_business AS (
    SELECT business_id, stars
    FROM business
    WHERE stars >= 4.5
),
review_aggregated AS (
    SELECT r.user_id, COUNT(r.review_id) AS total_reviews, AVG(b.stars) AS
avg_business_rating
```

```
    FROM review r
    INNER JOIN filtered_business b ON r.business_id = b.business_id
    GROUP BY r.user_id
)
SELECT u.name AS user_name,
       ra.total_reviews,
       ra.avg_business_rating
FROM users u
INNER JOIN review_aggregated ra ON u.user_id = ra.user_id
ORDER BY ra.total_reviews DESC
LIMIT 10);
```

# Non-relational Database

## Task 1: Common business categories

This query finds the top 10 most common business categories in the database. It leverages MongoDB's strengths in handling semi-structured data and performing transformations directly within the database, and using data pipeline for complex operations.

```
// Pipeline for finding most common categories
db.business.aggregate([
    // Split categories into an array
    { $project: { categories: { $split: ["$categories", ", "] } } },

    // Unwind the array
    { $unwind: "$categories" },

    // Count occurrences of each category
    { $group: { _id: "$categories", count: { $sum: 1 } } },

    // Sort and limit output
    { $sort: { count: -1 } },
    { $limit: 10 }
  ])
```

The query outputs with the following structure,

```
[
  {
    "_id": <category>,
    "count": <number of distinct business that uses this category>
  },
 {...}, ...
```

```
]
```

Adding an index on the categories field can speed up the initial `$split` and `$unwind` process if searches or partial filtering of categories are involved.

```
// Create index on categories
db.business.createIndex({ categories: "text" });

// Aggregation pipeline
db.business.aggregate([<aggregation pipeline>]);
```

Examining execution stats from the output using `.explain("executionStats")` we can see that the whole process of scanning and summing took 180 milliseconds with index, and 240 milliseconds without. Hence, adding index speeds up the process significantly.

## Task 2: Most reviewed businesses

This query finds the top 10 businesses with the most number of ratings in `yelp`, along with their ratings and other information if needed. Similar to the last example, this query also utilizes the aggregation pipeline of MongoDB, which is highly flexible and allows for efficient data transformations and computations. It also demonstrates how MongoDB handles cross-collection relationships in a NoSQL schema.

```
// Pipeline for counting reviews and join for business info
db.review.aggregate([
    // Count reviews for each business, sort in descending order
    { $group: { _id: "$business_id", review_count: { $sum: 1 } } },
    { $sort: { review_count: -1 } },
    { $limit: 10 },
    {
    // Join with business collection to get its info
      $lookup: {
        from: "business",
        localField: "_id",
        foreignField: "business_id",
        as: "business_info"
      }
    },
    { $unwind: "$business_info" }, // Flatten the business_info array
    {
      $project: {
        business_name: "$business_info.name",
        stars: "$business_info.stars",
        <extra info>,
        review_count: 1
      }
    }
  ]);
```

The query outputs with the following structure,

```
[
 {
    "_id": <business_id>,
    "review_count": <number of reviews>,
    "business_name": <business name>,
    "stars": <rating>
  },
 {...}, ...
]
```

Some ways of improving performance include: truncating number of documents or attributes processed in early steps, and adding index on searched attributes. In this case, `$limit` is applied after sorting, making sure no additional documents are dangling when performing join on `business`. We can also add indexes on `business_id` in `review` and `business` to optimize search time.

```
// Create index on business_id
```

```
db.review.createIndex({ business_id: 1 });
db.business.createIndex({ business_id: 1 }); //ignore if already exists

// Aggregation pipeline
db.review.aggregate([<aggregation pipeline>]);
```

Again, by examining execution stats we can see that without index it takes ~2900 milliseconds to process, whereas with index it only takes ~1700 milliseconds, almost halving the runtime.

# Tool Comparison

**General use:**

During the data setup, relational database Postgres requires schema design before importing the data, ensuring structured relationships and constraints. While a non-relational database MongoDB is easy to import the JSON files but it requires manual enforcement of constraints on primary keys and foreign keys for data consistency.

**Fitness, "Ergonomics", and performance:**

As a relational database, Postgres is good at working with structured data with strong relationships. It performs better in task 2 (identifying the top10 rated business) and task 3 (analyzing top10 high rating active users). Both tasks involve joining multiple tables, such as review, business, and users, which relies on structured, normalized data. PostgreSQL joins are more efficient by using foreign key constraints and indexing, ensuring fast lookups and consistent data integrity even for large datasets.

PostgreSQL is declarative allowing users to express complex tasks concisely. For example in task 3 (analyzing top10 high rating active users), PostgreSQL makes it straightforward to use `JOIN` to connect multiple tables, and `GROUP BY`, `COUNT` (counting reviews), `AVG` (averaging ratings), `ORDER BY` simplify aggregating and sorting data and easy to implement. While MongoDB requires `$lookup` for each join operation, such as between review and business, scanning and matching records across collections. These joins in MongoDB are less optimized in performance than PostgreSQL joins, leading to more complex queries pipelines.

MongoDB works best in semi-structured data as in task 1 (common business categories). In MongoDB, each business is stored as a single document with attributes like "categories" directly represented as an array within the document. This structure eliminates the need for normalization which is required in Postgres. Moreover, MongoDB's aggregation framework simplifies the process for splitting the categories string into individual entries. The `$project` stage uses the `$split` operator to transform the field into an array, and `$unwind` flattens the array for counting distinct categories. These operations are highly optimized as the pipeline handles semi-structured data directly within the database without requiring intermediate joins. This eliminates the need for complex preprocessing required in PostgreSQL, which makes MongoDB have more advantages in working with nested data structures.

# Reflections

For tasks that have been structured with clearly defined relationships, SQL system is a better choice for queries analytics, such as identifying top rated business (Task2) or top active high rating users (Task 3). Those requiring highly related datasets across different tables (even for large datasets), whose operations are well efficient as result of it, require schema designs that must be defined upfront and enforcing constraints (primary key, foreign keys, unique constraint) for data consistency. In addition, the declarative language used in SQL systems inherited the nature of describing the actions for results is more readable and user friendly for analysts and developers, while NoSQL systems might need more time to learn the JSON-like syntax and aggregation pipeline.

If data is semi-structured or nested structured, NoSQL system is a better candidate for queries analytics. Its flexibility in processing different types of files and no need for predefined schema design makes the data import process much more convenient. Sharding makes NoSQL system stand out in its scalability for tasks like distributed datasets. For example (Task1), a dataset with millions of `business` documents and frequent updates or queries by `categories` or `business_id` has better performance with NoSQL system as it is agile in distribution and process data across multiple nodes.