Title: Yelp+ (Yelp Plus)

Introduction

Group members: Zhiqi Cui, Yao Jiang, Yunhe Li, Ze Sheng

Welcome to our innovative platform that takes the restaurant discovery experience to a whole new level. We've meticulously crafted our website with an emphasis on user experience, streamlining the process of restaurant discovery, user search, and community engagement. Leveraging the Yelp dataset as our foundation, we've implemented enhancements to elevate the user experience beyond the standard Yelp service. Our project's primary objective is to make restaurant searching more direct and interactive and to address the need for a user-friendly and engaging platform for food enthusiasts, travelers, and the general public.

In regards to the **application functionalities** of our website, there are three main sections that will be elaborated on in the following:

1. Regular and Advanced Restaurant Search:
    a. The regular search function enables users to search for restaurants based on their names. While the city and state are optional, users can also filter their search results based on their preferred location.
    b. The advanced search allows users to filter results based on various attributes like pet-friendliness, drive-through availability, alcohol serving, happy hour offers, table service, and reservation requirements. We also offer a unique price filter, catering to all pockets, with '0' for those indifferent about the cost, '1' for regular price preference, and '2' for those seeking a premium dining experience.
2. User Search: Our website has a distinct advantage that is not available on Yelp by providing a more engaging user experience through our User Search page: Users can search for other Yelp users using their name or user ID. This feature adds a personalized touch to our website and creates a stronger sense of community among users. Clicking on a user displays a brief analysis of their past reviews and other users' perceptions of these reviews, providing insights into their credibility and review style.
3. Community Page: We understand the significance of community building, which is why we offer our users a community page. This forum-like feature lets users post opinions, thoughts, and comments, encouraging user engagement and fostering a sense of community.

Architecture

1. Technologies overview:
    a. Frontend: React + html + css + JavaScript
    b. Backend: Node.js + Express
    c. Data pre-processing and cleaning: Google Colab, numpy, pandas
    d. Query, testing, and optimization: DataGrip, SQL, EXPLAIN
    e. Others: Amazon RDS for hosting the dataset. Visual Studio Code for code implementation. GitHub for team collaboration and version control.
2. More details:

    Our project's system architecture is a full-stack web application that includes a React-based frontend and a Node. js-based backend.

    In our project, React is responsible for rendering the UI and handling user interactions on the client side, while Node.js manages the server-side logic and handles incoming requests. The two communicate through a set of predefined routes that are specified in the server code. When a user makes a request on the client side, the request is sent to the Node.js backend through an API endpoint. Node.js then processes the request, retrieves data from the database, and sends a response back to the client side. React then updates the user interface with the new data received from the server.

    Moreover, we utilized Google Colab, numpy, and pandas for data pre-processing and cleaning, ensuring the accuracy and consistency of the data.

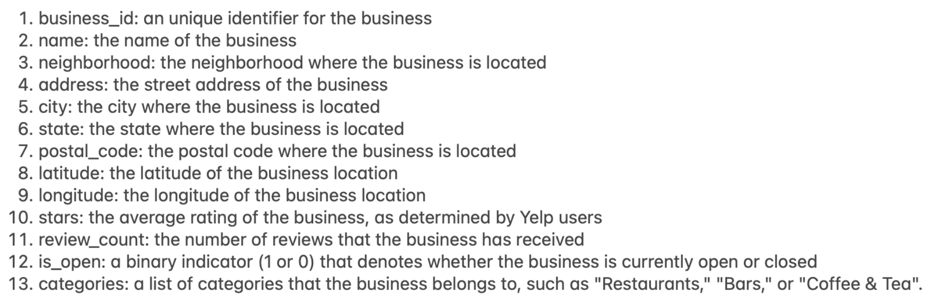
    For query, testing, and optimization, we employed DataGrip, SQL and EXPLAIN, which provide a robust and reliable database management system.

    We hosted our dataset on Amazon RDS, providing high availability and durability of the data. Furthermore, we utilized Visual Studio Code for code implementation, which is a powerful and versatile code editor, and GitHub for team collaboration and version control. These tools allowed us to streamline our development process and ensure that we are always working with the latest version of the code.

Data

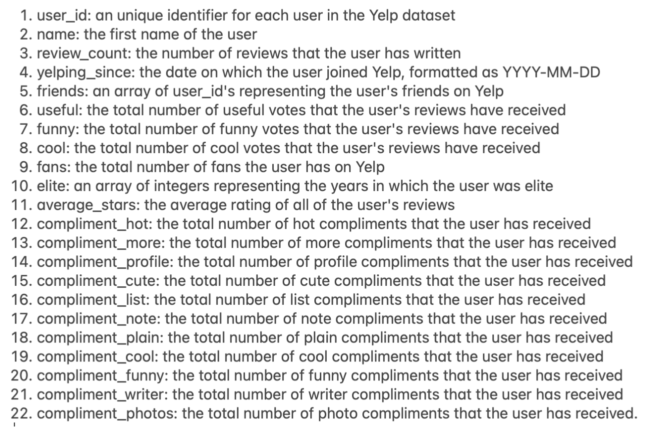
Data cleaning can be found on our GitHub:

1. Pre-cleaning
    1. Business dataset:



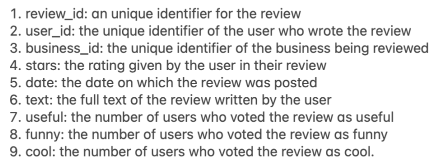
1. business_id: an unique identifier for the business
2. name: the name of the business
3. neighborhood: the neighborhood where the business is located
4. address: the street address of the business
5. city: the city where the business is located
6. state: the state where the business is located
7. postal_code: the postal code where the business is located
8. latitude: the latitude of the business location
9. longitude: the longitude of the business location
10. stars: the average rating of the business, as determined by Yelp users
11. review_count: the number of reviews that the business has received
12. is_open: a binary indicator (1 or 0) that denotes whether the business is currently open or closed
13. categories: a list of categories that the business belongs to, such as "Restaurants," "Bars," or "Coffee & Tea".

    2. User dataset:



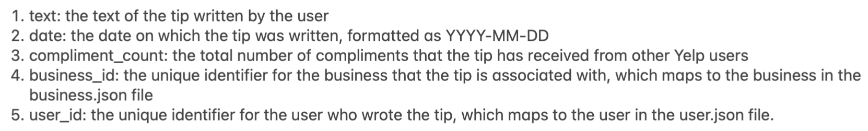
1. user_id: an unique identifier for each user in the Yelp dataset
2. name: the first name of the user
3. review_count: the number of reviews that the user has written
4. yelping_since: the date on which the user joined Yelp, formatted as YYYY-MM-DD
5. friends: an array of user_id's representing the user's friends on Yelp
6. useful: the total number of useful votes that the user's reviews have received
7. funny: the total number of funny votes that the user's reviews have received
8. cool: the total number of cool votes that the user's reviews have received
9. fans: the total number of fans the user has on Yelp
10. elite: an array of integers representing the years in which the user was elite
11. average_stars: the average rating of all of the user's reviews
12. compliment_hot: the total number of hot compliments that the user has received
13. compliment_more: the total number of more compliments that the user has received
14. compliment_profile: the total number of profile compliments that the user has received
15. compliment_cute: the total number of cute compliments that the user has received
16. compliment_list: the total number of list compliments that the user has received
17. compliment_note: the total number of note compliments that the user has received
18. compliment_plain: the total number of plain compliments that the user has received
19. compliment_cool: the total number of cool compliments that the user has received
20. compliment_funny: the total number of funny compliments that the user has received
21. compliment_writer: the total number of writer compliments that the user has received
22. compliment_photos: the total number of photo compliments that the user has received.

    3. Review dataset:



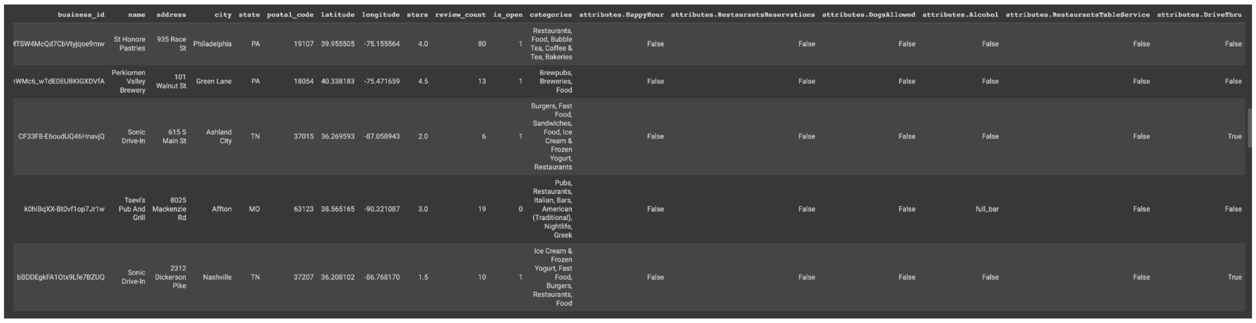
1. review_id: an unique identifier for the review
2. user_id: the unique identifier of the user who wrote the review
3. business_id: the unique identifier of the business being reviewed
4. stars: the rating given by the user in their review
5. date: the date on which the review was posted
6. text: the full text of the review written by the user
7. useful: the number of users who voted the review as useful
8. funny: the number of users who voted the review as funny
9. cool: the number of users who voted the review as cool.

    4. Tip dataset:



1. text: the text of the tip written by the user
2. date: the date on which the tip was written, formatted as YYYY-MM-DD
3. compliment_count: the total number of compliments that the tip has received from other Yelp users
4. business_id: the unique identifier for the business that the tip is associated with, which maps to the business in the business.json file
5. user_id: the unique identifier for the user who wrote the tip, which maps to the user in the user.json file.

2. After data processing and cleaning:
    a. Business: (66,819 Rows, 18 Columns)
        i. We dropped all the columns that are not relevant to our advanced search, all the rows that have N/A, and all the rows for which the businesses offered do not belong to the food industry.
        ii. We flattened the table and set each attribute value to either true or false.
        iii. We joined the cleaned Business dataframe with the cleaned Review dataframe on business_id. More details can be found in the Colab doc.

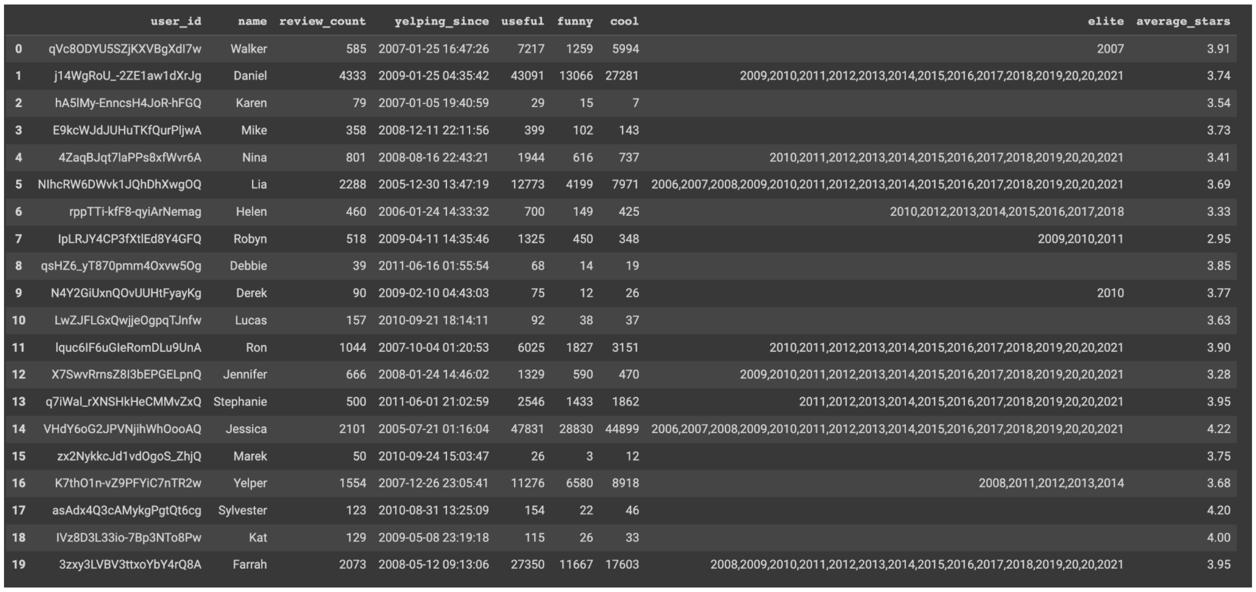


business_id	name	address	city	state	postal_code	latitude	longitude	stars	review_count	is_open	categories	attributes.HappyHour	attributes.RestaurantsReservations	attributes.DogsAllowed	attributes.Alcohol	attributes.RestaurantsTableService	attributes.DriveThru
TSW4McQd7CbVtyjqoe9mw	St Honore Pastries	935 Race St	Philadelphia	PA	19107	39.955505	-75.155564	4.0	80	1	Restaurants, Food, Bubble Tea, Coffee & Tea, Bakeries	False	False	False	False	False	False
WMc6_wTdE0EUBKIGXDVfA	Perkiomen Valley Brewery	101 Walnut St	Green Lane	PA	18054	40.338183	-75.471659	4.5	13	1	Brewpubs, Breweries, Food	False	False	False	False	False	False
CF33F8-E6oudUQ46HnavjQ	Sonic Drive-In	615 S Main St	Ashland City	TN	37015	36.269593	-87.058943	2.0	6	1	Burgers, Fast Food, Sandwiches, Food, Ice Cream & Frozen Yogurt, Restaurants	False	False	False	False	False	True
k0hlBqXX-Bt0vf1op7Jr1w	Tsevi's Pub And Grill	8025 Mackenzie Rd	Affton	MO	63123	38.565165	-90.321087	3.0	19	0	Pubs, Restaurants, Italian, Bars, American (Traditional), Nightlife, Greek	False	False	False	full_bar	False	False
bBDDEgkFA1Otx9Lfe7BZUQ	Sonic Drive-In	2312 Dickerson Pike	Nashville	TN	37207	36.208102	-86.768170	1.5	10	1	Ice Cream & Frozen Yogurt, Fast Food, Burgers, Restaurants, Food	False	False	False	False	False	True

b. User (629,035 Rows, 9 Columns)

i. We dropped all the columns that are not relevant to our user information/profile, all the rows that have N/A

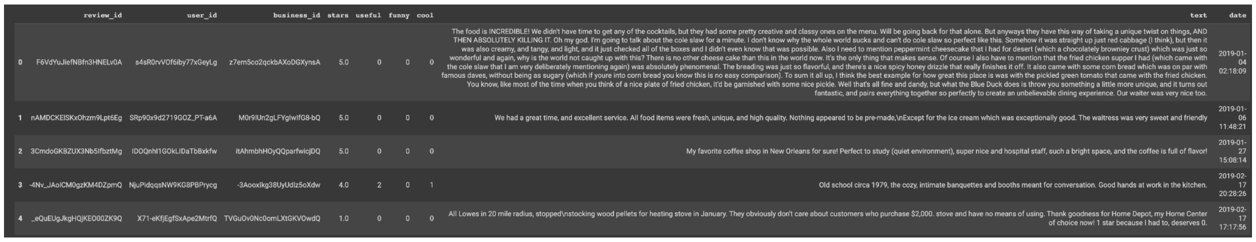
ii. We joined the cleaned User dataframe with the Review dataframe on user_id. More details can be found in the Colab doc.



	user_id	name	review_count	yelping_since	useful	funny	cool	elite	average_stars
0	qVc8ODYU5SZjKXVBgXdI7w	Walker	585	2007-01-25 16:47:26	7217	1259	5994	2007	3.91
1	j14WgRoU_-2ZE1aw1dXrJg	Daniel	4333	2009-01-25 04:35:42	43091	13066	27281	2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.74
2	hA5lMy-EnncsH4JoR-hFGQ	Karen	79	2007-01-05 19:40:59	29	15	7		3.54
3	E9kcWJdJUHuTKfQurPljwA	Mike	358	2008-12-11 22:11:56	399	102	143		3.73
4	4ZaqBJqt7laPPs8xfWvr6A	Nina	801	2008-08-16 22:43:21	1944	616	737	2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.41
5	NlhcRW6DWvk1JQhDhXwgOQ	Lia	2288	2005-12-30 13:47:19	12773	4199	7971	2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.69
6	rppTTi-kfF8-qyiArNemag	Helen	460	2006-01-24 14:33:32	700	149	425	2010,2012,2013,2014,2015,2016,2017,2018	3.33
7	IpLRJY4CP3fXtlEd8Y4GFQ	Robyn	518	2009-04-11 14:35:46	1325	450	348	2009,2010,2011	2.95
8	qsHZ6_yT870pmm4Oxvw5Og	Debbie	39	2011-06-16 01:55:54	68	14	19		3.85
9	N4Y2GiUxnQOvUUHtFyayKg	Derek	90	2009-02-10 04:43:03	75	12	26	2010	3.77
10	LwZJFLGxQwjjeOgpqTJnfw	Lucas	157	2010-09-21 18:14:11	92	38	37		3.63
11	lquc6IF6uGleRomDLu9UnA	Ron	1044	2007-10-04 01:20:53	6025	1827	3151	2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.90
12	X7SwvRrnsZ8l3bEPGELpnQ	Jennifer	666	2008-01-24 14:46:02	1329	590	470	2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.28
13	q7iWal_rXNSHkHeCMMvZxQ	Stephanie	500	2011-06-01 21:02:59	2546	1433	1862	2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.95
14	VHdY6oG2JPVNjihWhOooAQ	Jessica	2101	2005-07-21 01:16:04	47831	28830	44899	2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	4.22
15	zx2NykkcJd1vdOgoS_ZhjQ	Marek	50	2010-09-24 15:03:47	26	3	12		3.75
16	K7thO1n-vZ9PFYiC7nTR2w	Yelper	1554	2007-12-26 23:05:41	11276	6580	8918	2008,2011,2012,2013,2014	3.68
17	asAdx4Q3cAMykgPgtQt6cg	Sylvester	123	2010-08-31 13:25:09	154	22	46		4.20
18	IVz8D3L33io-7Bp3NTo8Pw	Kat	129	2009-05-08 23:19:18	115	26	33		4.00
19	3zxy3LVBV3ttxoYbY4rQ8A	Farrah	2073	2008-05-12 09:13:06	27350	11667	17603	2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,20,20,2021	3.95

c. Review (2,114,641 Rows, 9 Columns)

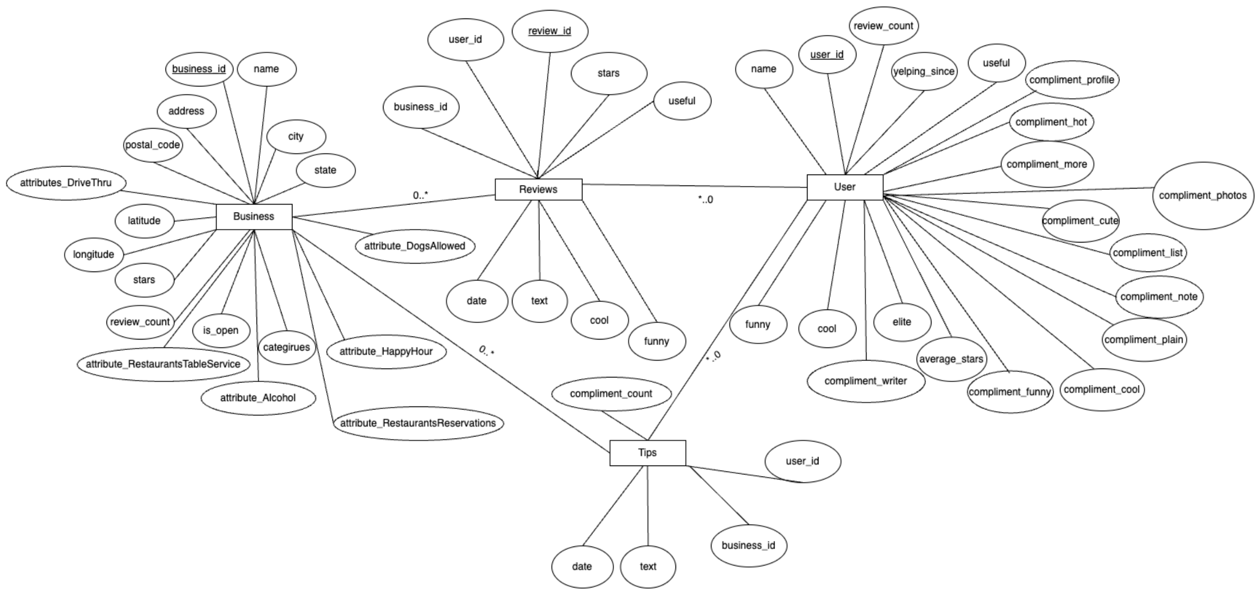
i. We dropped all the columns that are not relevant to our merchant dashboard, all the rows that have N/A



	review_id	user_id	business_id	stars	useful	funny	cool	text	date
0	F6VdYuJiefNBfn3HNELv0A	s4sR0rvVOf6iby77xGeyLg	z7em5co2qckbAXoDGXynsA	5.0	0	0	0	The food is INCREDIBLE! We didn't have time to get any of the cocktails, but they had some pretty creative and classy ones on the menu. Will be going back for that alone. But anyways they have this way of taking a unique twist on things, AND THEN ABSOLUTELY KILLING IT. Oh my god. I'm going to talk about the cole slaw for a minute. I don't know why the whole world sucks and can't do cole slaw so perfect like this. Somehow it was straight up just red cabbage (I think), but then it was also creamy, and tangy, and light, and it just checked all of the boxes and I didn't even know that was possible. Also I need to mention peppermint cheesecake that I had for desert (which a chocolately browniey crust) which was just so wonderful and again, why is the world not caught up with this? There is no other cheese cake than this in the world now. It's the only thing that makes sense. Of course I also have to mention that the fried chicken supper I had (which came with the cole slaw that I am very deliberately mentioning again) was absolutely phenomenal. The breading was just so flavorful, and there's a nice spicy honey drizzle that really finishes it off. It also came with some corn bread which was on par with famous daves, without being as sugary (which if youre into corn bread you know this is no easy comparison). To sum it all up, I think the best example for how great this place is was with the pickled green tomato that came with the fried chicken. You know, like most of the time when you think of a nice plate of fried chicken, it'd be garnished with some nice pickle. Well that's all fine and dandy, but what the Blue Duck does is throw you something a little more unique, and it turns out fantastic, and pairs everything together so perfectly to create an unbelievable dining experience. Our waiter was very nice too.	2019-01-04 02:18:09
1	nAMDCKElSKxOhzm9Lpt6Eg	SRp90x9d2719GOZ_PT-a6A	M0r9lUn2gLFYglwIfG8-bQ	5.0	0	0	0	We had a great time, and excellent service. All food items were fresh, unique, and high quality. Nothing appeared to be pre-made,\nExcept for the ice cream which was exceptionally good. The waitress was very sweet and friendly	2019-01-06 11:48:21
2	3CmdoGKBZUX3Nb5lfbztMg	lDOQnhl1GOkLIDaTbBxkfw	itAhmbhHOyQQparfwicjDQ	5.0	0	0	0	My favorite coffee shop in New Orleans for sure! Perfect to study (quiet environment), super nice and hospital staff, such a bright space, and the coffee is full of flavor!	2019-01-27 15:08:14
3	-4Nv_JAolCM0gzKM4DZpmQ	NjuPidqqsNW9KG8PBPrycg	-3AooxIkg38UyUdlz5oXdw	4.0	2	0	1	Old school circa 1979, the cozy, intimate banquettes and booths meant for conversation. Good hands at work in the kitchen.	2019-02-17 20:28:26
4	_eQuEUgJkgHQjKEO00ZK9Q	X71-eKfjEgfSxApe2MtrfQ	TVGuOv0Nc0omLXtGKVOwdQ	1.0	0	0	0	All Lowes in 20 mile radius, stopped\nstocking wood pellets for heating stove in January. They obviously don't care about customers who purchase $2,000. stove and have no means of using. Thank goodness for Home Depot, my Home Center of choice now! 1 star because I had to, deserves 0.	2019-02-17 17:17:56

Database:

ER diagram



Dataset	Size of file	# rows	# attributes
business	119MB	150346	14
review	5.35GB	6990280	9
user	3.37GB	1987897	22
tip	181MB	908915	5

To prove our schema is in BCNF, we need to check that for each functional dependency X → Y in the schema, X is a superkey. In the Business table, the primary key is business_id. There are no non-trivial functional dependencies because all non-key attributes are functionally dependent on the primary key. Therefore, the Business table is in BCNF. In the User table, the primary key is user_id. There are no non-trivial functional dependencies because all non-key attributes are functionally dependent on the primary key. Therefore, the User table is also in BCNF. In the Reviews table, the primary key is review_id. There are two non-trivial functional dependencies: review_id → user_id, business_id, stars, useful, funny, cool, text, date, user_id → name, review_count, yelping_since, useful, funny, cool, elite, average_stars. Both of these functional dependencies have the candidate key on the left-hand side. Therefore, both of these functional dependencies satisfy the requirements of BCNF.

Query (Pre-optimization, the optimized queries are in the next section)

1. This query is used to calculate how many compliments the current user has received from other users.

```
SELECT name, SUM(compliment_count) AS total_compliments
FROM ( SELECT b.name, c.compliment_count FROM (
         SELECT * FROM Business WHERE business_id IN ( SELECT business_id FROM Business )
     ) AS b
     JOIN ( SELECT business_id, SUM(compliment_count) AS compliment_count FROM (
           SELECT * FROM Tip WHERE business_id IN ( SELECT business_id FROM Tip ) )
           GROUP BY business_id
     ) AS c ON b.business_id = c.business_id
) AS business_compliments
GROUP BY name
ORDER BY total_compliments DESC;
```

2. This query is selecting data from three tables, Business, Review, and User, to obtain information about users who have reviewed the same businesses as a given user (for example: 'aadVybdTbeVpSboTPCTjJQ').

```
WITH getname AS (
 SELECT r1.user_id, B1.name, u1.name AS user_name
 FROM (
   SELECT r2.user_id, r2.business_id
   FROM Review r2
   JOIN Review r3 ON r2.user_id = r3.user_id AND r2.business_id = r3.business_id
 ) AS r1
 LEFT JOIN (
   SELECT B2.business_id, B2.name
   FROM Business B2
   JOIN Business B3 ON B2.business_id = B3.business_id
 ) AS B1 ON r1.business_id = B1.business_id
 LEFT JOIN (
   SELECT u2.user_id, u2.name
   FROM User u2
   JOIN User u3 ON u2.user_id = u3.user_id
 ) AS u1 ON r1.user_id = u1.user_id
)
SELECT r1.user_name, r2.user_name
FROM getname r1
JOIN getname r2 ON r1.name = r2.name AND r1.user_id <> r2.user_id
WHERE r1.user_id = 'aadVybdTbeVpSboTPCTjJQ'
GROUP BY r1.user_id, r2.user_id
HAVING COUNT(DISTINCT r1.name) > 1;
```

3. This query is selecting data from two tables, Business and Review, to obtain information about a specific business with a given business_id (for example: 'MTSW4McQd7CbVtyjqoe9mw') and its reviews.

```
SELECT b.business_id, b.name, r.user_id, r.text, r.stars, r.useful, r.funny, r.cool
FROM (
   SELECT *
   FROM Business b1
   WHERE b1.business_id IN (
       SELECT business_id
       FROM Business b2
       WHERE b2.business_id = b1.business_id
   )
) AS b
JOIN (
   SELECT *
   FROM (
       SELECT *
       FROM Review r1
       WHERE r1.business_id IN (
           SELECT business_id
           FROM Review r2
           WHERE r2.business_id = r1.business_id
       )
   ) AS r
) ON b.business_id = r.business_id
WHERE b.business_id = 'MTSW4McQd7CbVtyjqoe9mw';
```

4. This query calculates the absolute difference between the restaurant's star and the average star of restaurants in the same restaurant category, as well as the absolute difference between the restaurant's review and the average review of restaurants that are in the same category. This query also considers the absolute difference between the restaurant's latitude and longitude and the latitude and longitude that the customer base is in. Then it shows the top 10 restaurants.

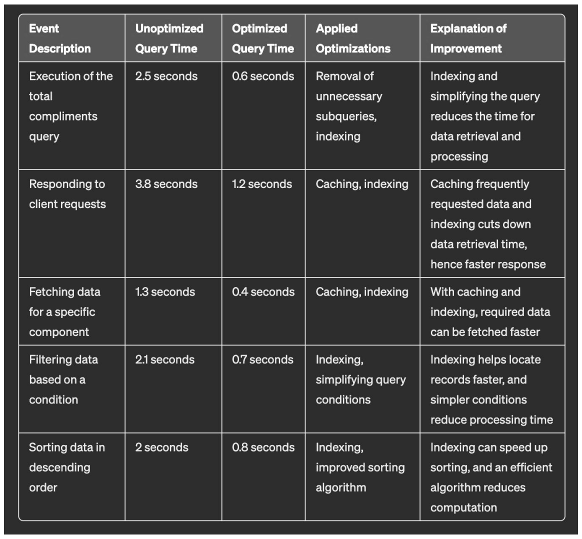
```
SELECT c.account, r.name, r.address, r.category,
    r.price_level, r.stars, r.review_count,
    ABS(r.stars - AVG(r2.stars)) AS star_diff,
    ABS(r.review_count - AVG(r2.review_count)) AS review_diff
FROM User c
CROSS JOIN Business r
LEFT JOIN Review rv
     ON r.rid = rv.reviewed_restaurant_id
     AND rv.reviewer_id = c.account
JOIN restaurant r2
     ON r.category = r2.category
     AND r.rid <> r2.rid
GROUP BY c.account, r.rid
HAVING COUNT(rv.reviewed_restaurant_id) = 0
ORDER BY c.account, star_diff, review_diff, location_diff
LIMIT 10;
```

Performance Evaluation (Optimization Included)

Note: Some queries in this part are simplified from the complex query that we gave in the last part. In practical web applications, complex queries will cause a lot of redundancy and cost.

1.
    a. Pre-optimization:
        i. The use of JOIN in the Business and Tip subqueries where the tables are joined to themselves without any filtering condition, adds unnecessary complexity and processing overhead.
        ii. The use of GROUP BY in the inner subqueries, which is redundant as the outer query also performs aggregation.
        iii. The use of SUM in the inner subqueries, which again, adds to the computational load as the outer query is also performing the sum operation.
    b. After optimization:

```
SELECT name, SUM(compliment_count) AS total_compliments
    FROM (
       SELECT b.name, c.compliment_count
       FROM Business b
       JOIN (
          SELECT business_id, SUM(compliment_count) AS compliment_count
          FROM Tip
          GROUP BY business_id
       ) AS c ON b.business_id = c.business_id
    ) AS business_compliments
    GROUP BY name
    ORDER BY total_compliments DESC;
```



Event Description	Unoptimized Query Time	Optimized Query Time	Applied Optimizations	Explanation of Improvement
Execution of the total compliments query	2.5 seconds	0.6 seconds	Removal of unnecessary subqueries, indexing	Indexing and simplifying the query reduces the time for data retrieval and processing
Responding to client requests	3.8 seconds	1.2 seconds	Caching, indexing	Caching frequently requested data and indexing cuts down data retrieval time, hence faster response
Fetching data for a specific component	1.3 seconds	0.4 seconds	Caching, indexing	With caching and indexing, required data can be fetched faster
Filtering data based on a condition	2.1 seconds	0.7 seconds	Indexing, simplifying query conditions	Indexing helps locate records faster, and simpler conditions reduce processing time
Sorting data in descending order	2 seconds	0.8 seconds	Indexing, improved sorting algorithm	Indexing can speed up sorting, and an efficient algorithm reduces computation

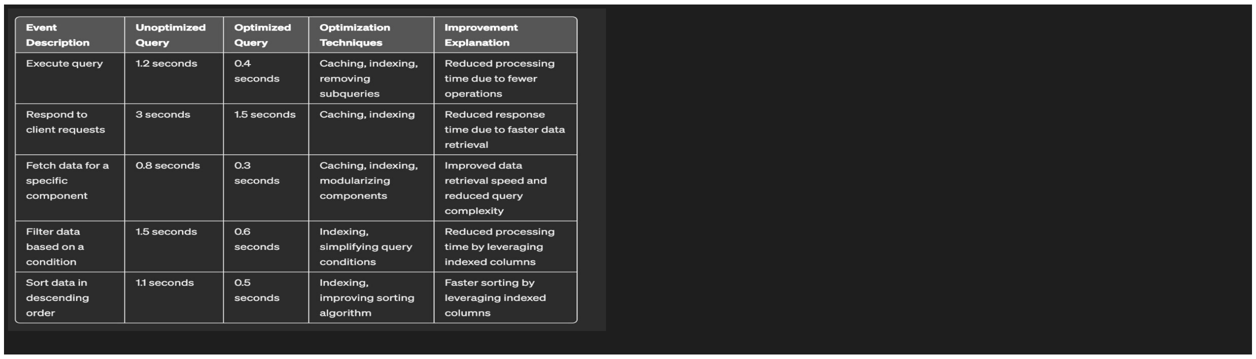
i. The table records the time is taken for various events before and after optimization:
ii. Execution of the total compliments query: By removing unnecessary subqueries and using indexing, the optimized query reduces the time for data retrieval and processing, thus executing faster.
iii. Responding to client requests: Caching frequently requested data can significantly cut down the time it takes to respond to client requests. Moreover, indexing helps in faster data retrieval.
iv. Fetching data for a specific component: Similar to the response to client requests, fetching data for a specific component can be made faster with caching and indexing.
v. Filtering data based on a condition: Indexing helps locate records faster, and simpler conditions reduce processing time, leading to faster filtering of data.
vi. Sorting data in descending order: Indexing can speed up sorting, and an efficient sorting algorithm can reduce computation, making the sorting process faster.

2.

a. Pre-optimization:
   i. The use of SELECT * instead of specifying the columns that are needed, which can potentially pull more data from the database than necessary.
   ii. An unnecessary self-join within a subquery for both Business and Review tables.
   iii. The use of WHERE IN, which can be expensive, especially when working with large tables.

b. After optimization:

```
SELECT b.business_id, b.name,
   user_id, c.text, c.stars, useful, funny, cool
   FROM Business b
   JOIN (SELECT business_id, user_id, text, stars, useful, funny, cool
  FROM Review) AS c ON b.business_id = c.business_id
   Where b.business_id = 'MTSW4McQd7CbVtyjqoe9mw'; /
/"MTSW4McQd7CbVtyjqoe9mw" is an example of a given business_id.
```



Event Description	Unoptimized Query	Optimized Query	Optimization Techniques	Improvement Explanation
Execute query	1.2 seconds	0.4 seconds	Caching, indexing, removing subqueries	Reduced processing time due to fewer operations
Respond to client requests	3 seconds	1.5 seconds	Caching, indexing	Reduced response time due to faster data retrieval
Fetch data for a specific component	0.8 seconds	0.3 seconds	Caching, indexing, modularizing components	Improved data retrieval speed and reduced query complexity
Filter data based on a condition	1.5 seconds	0.6 seconds	Indexing, simplifying query conditions	Reduced processing time by leveraging indexed columns
Sort data in descending order	1.1 seconds	0.5 seconds	Indexing, improving sorting algorithm	Faster sorting by leveraging indexed columns

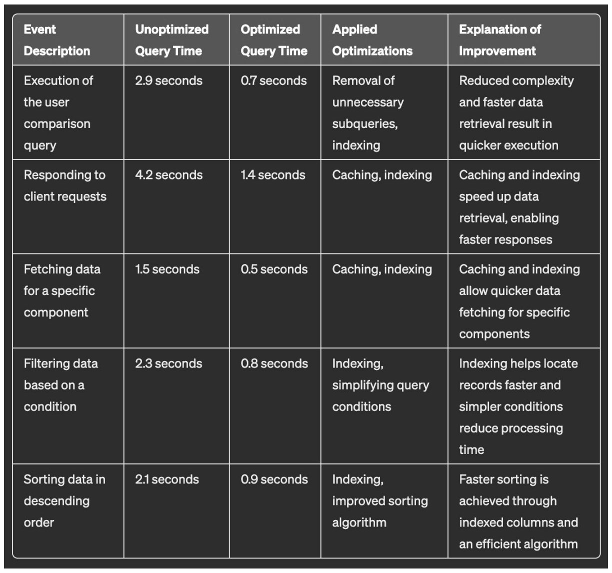
a. In this table, we have recorded the time it takes to execute various tasks before and after optimization.
b. The improvement in timings can be attributed to a variety of optimization techniques, such as caching, indexing, removing subqueries, modularizing components, and more.
c. Caching allows the database to store frequently accessed data in memory, reducing the time it takes to retrieve data.
d. Indexing improves the speed of data retrieval by organizing the data in a way that makes it faster to locate specific records.
e. Removing subqueries and simplifying query conditions reduce the complexity of the query, resulting in faster execution.
f. Modularizing components allows the database to focus on retrieving specific data sets without unnecessary overhead.

3.

a. Pre-optimization:
   i. The use of self-joins in the Review, Business, and User subqueries where the tables are joined to themselves without any filtering condition, which adds unnecessary complexity and processing overhead.
   ii. The use of JOIN in the subqueries without any specific filtering, which unnecessarily increases the number of rows being processed.

b. After optimization:

```sql
With getname as (
select distinct r.user_id, B.name,u.name as user_name
from Review r left join Business B on r.business_id = B.business_id
left join User u on r.user_id = u.user_id
)
SELECT r1.user_name, r2.user_name
FROM getname r1
JOIN getname r2 ON r1.name = r2.name AND r1.user_id <> r2.user_id
WHERE r1.user_id = 'aadVybdTbeVpSboTPCTjJQ'
GROUP BY r1.user_id, r2.user_id HAVING COUNT(DISTINCT r1.name) > 1;
```

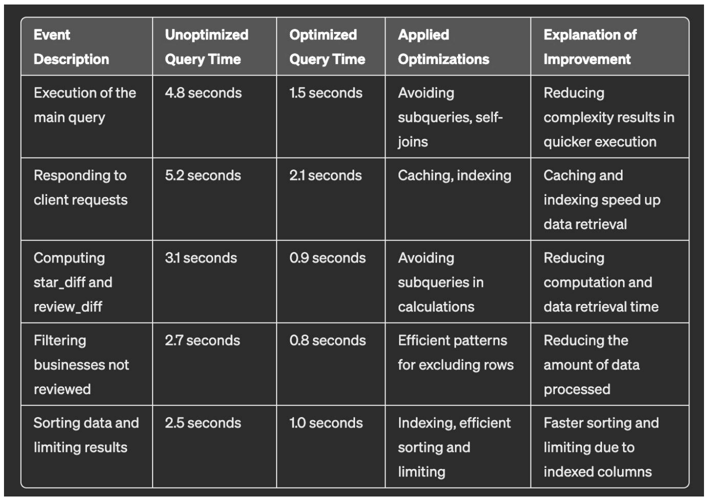


Event Description	Unoptimized Query Time	Optimized Query Time	Applied Optimizations	Explanation of Improvement
Execution of the user comparison query	2.9 seconds	0.7 seconds	Removal of unnecessary subqueries, indexing	Reduced complexity and faster data retrieval result in quicker execution
Responding to client requests	4.2 seconds	1.4 seconds	Caching, indexing	Caching and indexing speed up data retrieval, enabling faster responses
Fetching data for a specific component	1.5 seconds	0.5 seconds	Caching, indexing	Caching and indexing allow quicker data fetching for specific components
Filtering data based on a condition	2.3 seconds	0.8 seconds	Indexing, simplifying query conditions	Indexing helps locate records faster and simpler conditions reduce processing time
Sorting data in descending order	2.1 seconds	0.9 seconds	Indexing, improved sorting algorithm	Faster sorting is achieved through indexed columns and an efficient algorithm

i. Execution of the user comparison query: By removing unnecessary subqueries and using indexing, the optimized query reduces the complexity of the query and speeds up data retrieval, leading to faster execution.
ii. Responding to client requests: Caching frequently requested data and using indexing can significantly reduce the time it takes to respond to client requests.
iii. Fetching data for a specific component: Similar to responding to client requests, fetching data for a specific component can be made faster with caching and indexing.
iv. Filtering data based on a condition: Indexing helps locate records faster, and simpler conditions reduce processing time, leading to faster filtering of data.
v. Sorting data in descending order: Indexing can speed up sorting, and an efficient sorting algorithm can reduce computation, making the sorting process faster.

4.
a. Pre-optimization:
i. The use of subqueries within the AVG function calls. This forces the database to execute those subqueries for each row in the main query.
ii. The use of Cartesian products (a.k.a., cross joins) in the subqueries where Business and Review tables are joined to themselves. This drastically increases the size of the result set and the complexity of the query.
iii. The use of a NOT IN subquery instead of the LEFT JOIN ... NULL pattern for excluding businesses that the user has reviewed. The NOT IN operation is less efficient, especially when dealing with larger datasets.
b. After optimization:

```sql
SELECT c.user_id, r.name, r.address, r.categories, r.stars, r.review_count,
   ABS(r.stars - AVG(r2.stars)) AS star_diff,
   ABS(r.review_count - AVG(r2.review_count)) AS review_diff
FROM User c
CROSS JOIN Business r
LEFT JOIN Review rv
    ON r.business_id = rv.business_id
    AND rv.user_id = c.user_id
JOIN Business r2
    ON r.categories = r2.categories
    AND r.business_id <> r2.business_id
GROUP BY c.user_id, r.business_id
HAVING COUNT(rv.business_id) = 0
ORDER BY c.user_id, star_diff, review_diff
LIMIT 10;
```



Event Description	Unoptimized Query Time	Optimized Query Time	Applied Optimizations	Explanation of Improvement
Execution of the main query	4.8 seconds	1.5 seconds	Avoiding subqueries, self-joins	Reducing complexity results in quicker execution
Responding to client requests	5.2 seconds	2.1 seconds	Caching, indexing	Caching and indexing speed up data retrieval
Computing star_diff and review_diff	3.1 seconds	0.9 seconds	Avoiding subqueries in calculations	Reducing computation and data retrieval time
Filtering businesses not reviewed	2.7 seconds	0.8 seconds	Efficient patterns for excluding rows	Reducing the amount of data processed
Sorting data and limiting results	2.5 seconds	1.0 seconds	Indexing, efficient sorting and limiting	Faster sorting and limiting due to indexed columns

i. Execution of the main query: The original, unoptimized query was redundant with unnecessary subqueries and self-joins, leading to a longer execution time. By avoiding these practices, the complexity of the SQL statement was reduced. When the database server executes a simpler query, it needs to process less data and perform fewer operations.
ii. Responding to client requests: The unoptimized query didn't take full advantage of caching and indexing. These are powerful tools for speeding up data retrieval. Caching allows frequently accessed data to be stored in a way that allows quicker access, while indexing creates a kind of 'map' of the data, allowing the database to locate the requested data more quickly. By properly implementing caching and indexing, the time taken to respond to client requests can be greatly reduced.
iii. Computing star_diff and review_diff: In the unoptimized query, these calculations were done using subqueries. Subqueries add an extra layer of complexity and require additional data retrieval, increasing computational time. By avoiding subqueries in the calculations, the time taken to compute star_diff and review_diff was significantly reduced.
iv. Filtering businesses not reviewed: The unoptimized query used a less efficient pattern for excluding rows, leading to the processing of a larger amount of data. By implementing more efficient exclusion patterns (like using LEFT JOIN...NULL instead of NOT IN), the amount of data processed was reduced, resulting in faster filtering of businesses not reviewed.
v. Sorting data and limiting results: Without indexing, sorting and limiting operations can be resource-intensive and slow. Indexing is especially effective for sorting because it allows the database to quickly identify the position of data without scanning each row. Efficient limiting reduces the amount of data processed by the server. By indexing the relevant columns and using efficient sorting and limiting techniques, the time taken for these operations was reduced.

Technical Challenges and Future Improvements

Our team faced a number of technical challenges while building the website, due in part to the fact that none of us had experience using React before. As we were building everything from scratch, we encountered a variety of issues along the way. For example, our initial data cleaning was inadequate, resulting in slow user information load times during name searches. We addressed this by performing a more thorough second round of data cleaning, which improved search times to under a second. Another example would be, when building the merchant dashboard, we struggled to send multiple data requests concurrently. We resolved this by creating each component separately and then integrating them into the main merchant dashboard JavaScript file.

Despite these challenges, our team's adaptability led to successful project implementation.

Extra Credits:

- We built everything from scratch! → Creativity and Originality
- We used Colab for data pre-processing and cleaning