NextNest

Introduction:

Moving to a new place can be a daunting decision, as there are several factors to consider such as income, weather, and safety. Our web application aims to simplify this process by providing a centralized platform where users can easily access and compare information on demographics, wages, cost of living, crime, and other crucial details. Our team comprises of three members, Jack Shumway <jshumway@seas.upenn.edu>, Ankita Patel <ankip@seas.upenn.edu>, Divya Palani <divya24@seas.upenn.edu> and we were motivated to create this app to make it easier for individuals to make informed decisions about their next move or trip.

Web App Description:

Our web application consists of 9 main pages - the Landing Page, Login Page, Search Page, Details Page, Compare Page, Trends Page, Heatmap Page, Review Page, and Profile Page.

Here's a brief overview of the pages and functionality our app offers:

- **Landing Page** - Provides a brief overview of the app's functionality.
- **Login Page** - Allows users to create an account or log in using their Google or GitHub accounts (powered by Auth 0).
- **Search Page** - Enables users to filter and search for locations based on specific metrics and criteria. Results can be sorted and are paginated.
- **Details Page** - Provides comprehensive information on a particular location, including demographic data, quality of life, education, cost of living, income, and weather.
- **Compare Page** - Enables users to select multiple geographies and compare them side by side based on various metrics.
- **Trends Page** - Enables users to view trends for specific metrics in a geographic area over a selected timeframe.
- **Heatmap Page** - Provides an interactive map that enables users to view specific metrics for different geographies at different levels, such as states, metros, cities/towns, and counties overlaid on a map of the united states.
- **Review Page** - Allows users to rate and write reviews of locations they have lived in or visited.
- **Profile Page** - Allows users to view their profile information, favorited places and see past reviews they have left. Users can also edit/delete their past reviews.

Overall, our web application provides a one-stop-shop for individuals looking to make informed decisions about finding their perfect destination, saving them time and effort in conducting in-depth research across various websites.

Architecture:

List of Technologies: MySQL, AWS, REACTJS, NODEJS/Express, PYTHON, Recharts, Material UI, React Leaflet

Description of system architecture/application: The architecture of our application is straightforward. The client connects to a server which serves our compiled application and any static assets. We have a separate backend server which handles api requests which runs Express/Nodejs. The backend server connects to an RDS mysql instance hosted in AWS and relays queries and data.

Data:

Data Source	File Size	Format	Records	Link
Census Community Survey Data	~100 MB	API	408,023	https://data.census.gov/
Census Geographic Shapes	~24 MB	Shape Files	36,120	https://www2.census.gov/geo/
FBI Crime Data	~1 MB	CSV	11,307	https://ucr.fbi.gov/crime-in-the-u.s/2016/crime-in-the-u.s.-2016
EPI Cost of Living Data	~8 MB	CSV	38,012	https://www.epi.org/resources/budget/budget-map/

DataSet 1 - Census Survey Data
Description: Data from the Census Community Survey (at the state, county, metro and place level). This dataset includes information on populations, demographics, income, education and several other useful metrics for the entire United States. We pulled select field via the census api for all available years (2009-2021)

Usage: This is the primary dataset used for displaying metrics about geographic

locations. Nearly every page depends on this dataset.

DataSet 2 - Census Geospatial Data

Description: This dataset is geospatial data published by the census, which allows for plotting geographic census areas on a map. We have pulled the geographies of states, counties, metros and places to allow users to visualize at different levels. The Census Bureau provides this data in a binary file format (shape files), which was downloaded from an FTP site, processed, and converted to a geojson format that could be stored in a database.

Usage: The geographic shape files were used to power our interactive map. They are pulled and rendered by leaflet after applying some shading logic to capture metric levels. We also wrote a geo json to SVG converter to be able to use these geometries to render icons (for example on the details and comparison pages).

DataSet 3 - FBI Crime Data

Description: This dataset captures crime in the United States at the county and city level as reported by local and federal law enforcement agencies.

Usage: The initial data which had counts of specific crimes was aggregated to get counts of violent crimes, which was then used to build a crime index and a crimes per 100k metric. We joined this to our census data to provide additional insights into crime for a specific geographic area.

DataSet 4 - Economic Policy Institute Cost of Living Data

Description: This dataset captures cost of living for counties and select metro areas.

Usage: The cost of living data was used to build a cost of living index as well as an annual cost number. These data points were merged with our census data to give users the ability to consider living expenses when exploring our site.

Relationships:

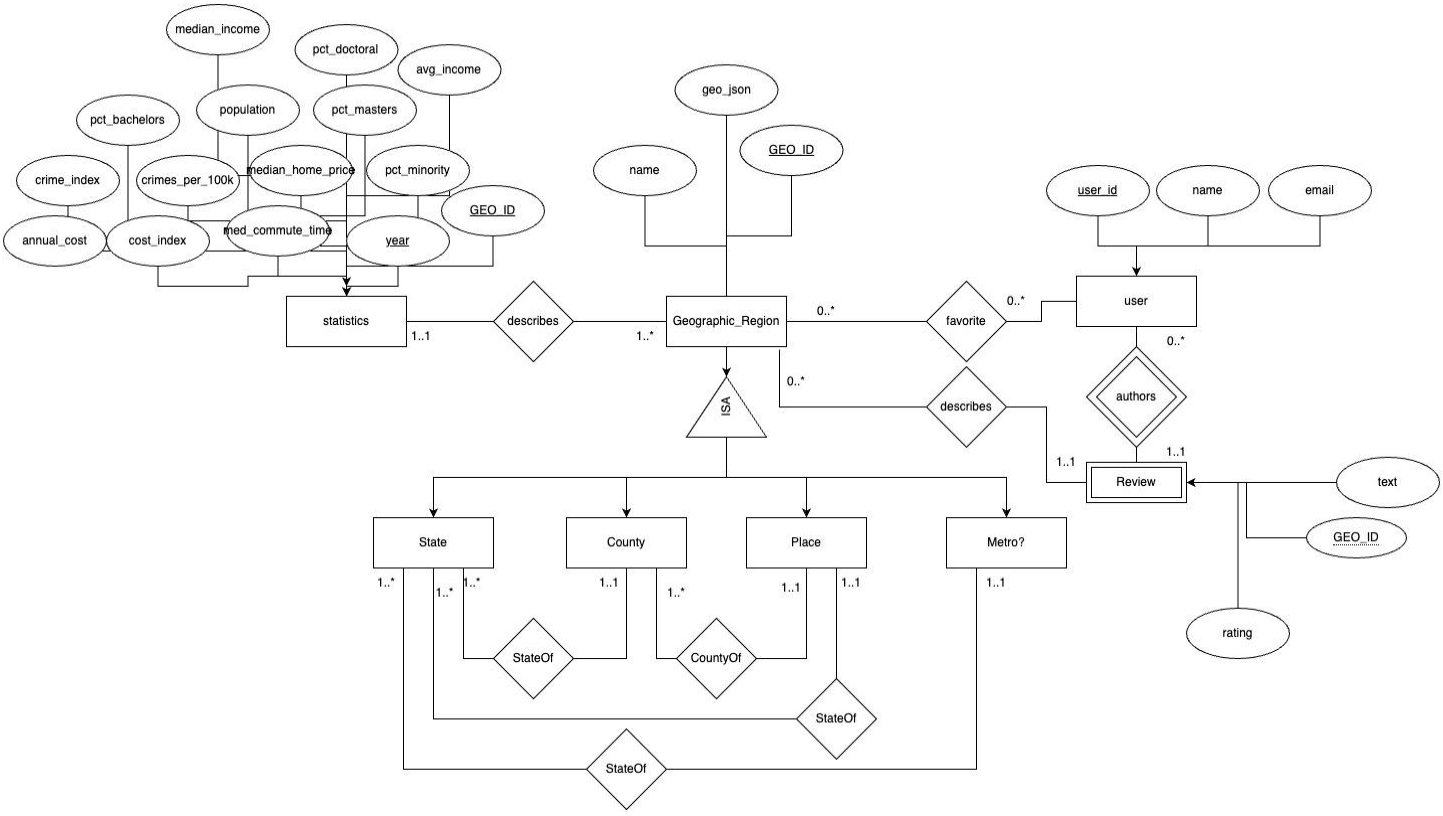
The data is all related by some reference to geographic places. The geospatial data defines the perimeter of a geographic place. The census data provides a wide variety of statistical information about a place. The crime data and cost of living data provide supplementary data about places. As a result the data is joined together based on a geographic identifier.

Database:

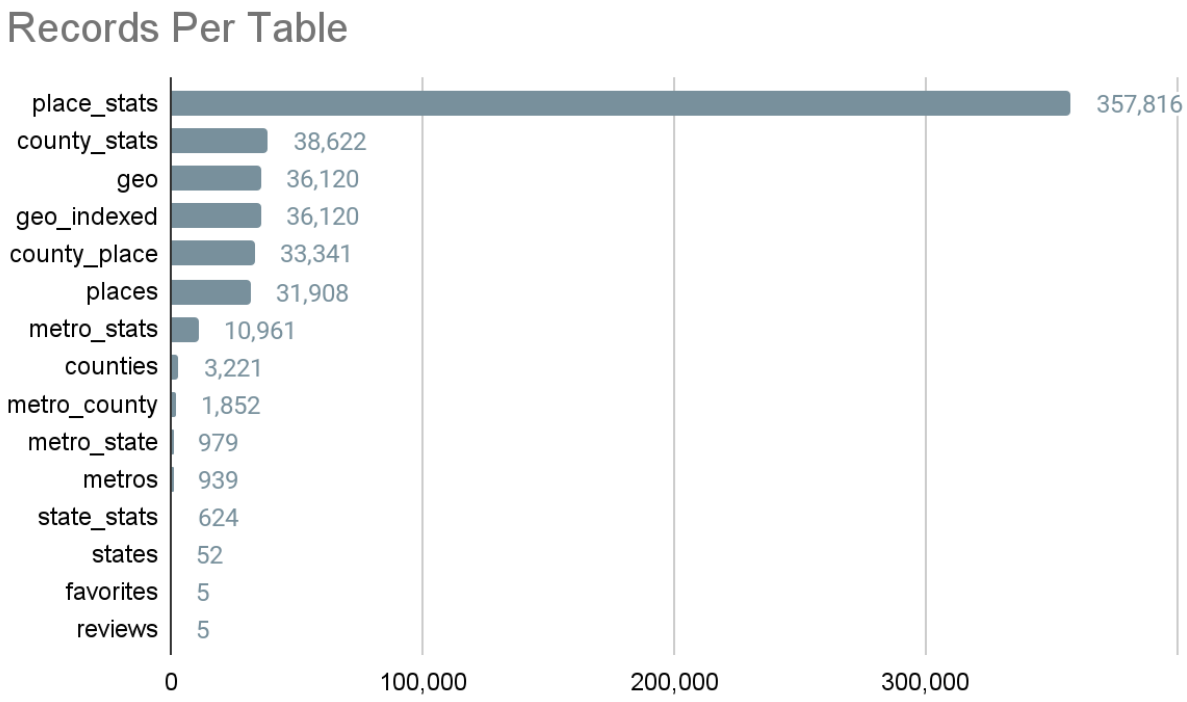
Data Ingestion Explanation: After data cleansing and table creation in the database, we loaded data both manually through datagrip as well as through a python script.

Entity Resolution Efforts: Entity resolution was simple between the census data and the geospatial data as there were common census provided keys for the places. Merging the crime data and cost data to the census data was much more involved and required name cleansing and fuzzy matching on the names of places. This required several steps such as matching on a few common fields (such as state), cleaning the names (removing common words and special characters), calculating a similarity score, and determining an appropriate threshold. As a result some unresolved records had to be discarded.

ER Diagram:



Number of Instances in Each Table:



Normal Form: 3NF and BCNF

Justification: There is no duplication of data and hence, there is no data redundancy or inconsistency in our tables. There are also no partial/transitive dependencies. All of our tables have a single candidate key (generally geo_id, or geo_id paired with another field). This means that for every functional dependency A→B, A is always a superkey in our schema.

API Specification: See Appendix A

Queries: See Appendix B

Performance Evaluation:

We used various techniques including virtualization, server side caching, pagination, indexing and query optimization to enhance performance of our web application. The type of the technique we used for a specific event was carefully selected based on the nature of the application, functionality that the specific event was providing, type and nature of the query results. We have used the network performance in the client side development window to record the running time of various events. We have measured the running time by hard refreshing and removing the cache over several runs for each specific event. Summary of optimizations used throughout our web application is summarized below:

Heatmap Page - Since this page provides an interactive map, which loads a lot of data,we have had issues loading the page in a timely manner. We have employed filtering and added ***geo spatial index*** to improve the performance by reducing the load time 50%. Details of unoptimized query and methodology used for optimization as well as the optimized query for this page is provided in Attachment 2 under **geo_shapes** API.

Search Page - We employed ***pagination*** to optimize the performance for **search** API. Since pagination only renders a specific number of results at a time, the performance is enhanced by partitioning big result sets into smaller sets. Refer to the search API in Attachment 2 for details.

Details Page - We have used ***indexing*** on the user for the **favorite** and **review** APIs. Since the user_index allows to efficiently retrieve the user instead of scanning the entire favorite table, it is expected to improve performance when the application has several users.

Compare Page - This page enables users to select multiple geographies. We needed to get names of various geographies as options. There are over 30,000 entries combined for names, it took about 2 seconds to fetch all names. Since the geography names are static, we employed server side caching using node cache. We originally considered materialized view for optimization as materialized views are suited for fetching static information.However, the AWS RDS database used for this project does not support materialized views, we opted to use ***in-memory***

caching instead**.** Refer to **geo_names** API for more details. Since server side caching fetches information from the cached memory instead of pulling the requested data from the database(once it is cached in the memory), the results are returned back much faster.

Trends Page - We carefully evaluated our query for the geo_trends API and optimized the performance by selecting the smaller relation as the outer relation while performing the join. Refer to **geo_trends** API for more details.

Technical Challenges:

Geometric Shape Conversion and Compression

Description: The geometric shapes from the census FTP site were stored as binary shape files which were not compatible with front-end development or database storage. Once we converted the files to JSON format, they were too large to store and fetch efficiently.

Resolution: Using python libraries, we had to convert the files from shape files to geo json files. We then wrote a script to reduce the amount of detail stored in the geometric shapes so that the size was manageable to send over the network. This was necessary as the user often needs to select all geometric shapes to display them on the map. Even after the size reduction, some views of the map were still sluggish. To further optimize we moved to only fetching shapes that were in the map viewport.

Map shading

Description: To make the map intuitive and useful, we needed to shade it in such a way that it was easy to interpret. Raw values did not work for this and outliers further complicated the issue.

Resolution: In order to get good shading we tried several approaches. The approach we ended up using was a version of min-max scaling. We fetched the values in sorted order and used the formula (value - min / max - min) to calculate the opacity of the elements. Even with this approach outliers caused a lot of problems (for example the population of a few large states such as California and Texas would cause all other states to be very lightly shaded). As a result we removed outliers from the calculation by throwing out highest and lowest percentiles.

Metric Formatting/Display

Description: The wide variety of metrics we have available made it difficult to format appropriately. It also made rendering of input elements (such as filters) difficult.

Resolution: We ended up creating a large mapping of metrics to formatters and renderers in order to properly display each item. This gave us a centralized way to create a consistent layout across our site.

List Rendering Lag

Description: In many cases we had to fetch a relatively large number of records to display, such as to populate dropdown lists and display search results. The large number of records made for slow fetching times and caused rendering issues and page timeouts.

Resolution: For dropdowns we were able to use virtualized lists to make them much more performant. Virtualized lists only render a window of the data and adjust as a user scrolls or searches. In the case of search results we moved to a paginated layout which allowed us to fetch fewer records and made the display more performant. It also allowed us to amortize the networking costs over several requests.

Interesting Findings:

One thing that was very interesting was learning about how geospatial data works in mysql. While you can store geo json in mysql, it is stored simply as json data and there is no way to do geospatial operations on it without first converting it. Mysql supports well-known-text and well-known-binary format for geospatial data, so we added a column with our geo json converted to the binary format. With that conversion we were able to build a spatial index on the column and take advantage of geospatial functions. Mysql uses something called an R-tree to build the index which allows for quick lookups of overlapping shapes. This was helpful for quickly finding shapes that were inside the viewport of our map and helped with performance.

Extra Credit Features:

Our web application features two bonus functionalities that contribute to a smoother user experience. Firstly, we have integrated user login experience with Google and GitHub, allowing users to conveniently sign in using their existing accounts. Secondly, We integrated the Visual Crossing Weather API to fetch real-time weather data for locations in the detail page. These additional features not only enhance the functionality of our application but also offer added convenience to our users.

We have also implemented server side caching using node cache and used virtualization to improve performance. We also employed an autocomplete feature for several of our web application pages that completes the name of the geographical place for the user.

Appendix:

Exhibit A - API Specification:

Route: /review
Description: Creates a new review/Updates an existing review
Request Method: POST
Route Parameter(s): None
Query Parameter(s): None
Request Body: { user: <user_id>, lived: <bool>, geo_type: <geo_type>, geo_id: <geo_id>, stars: <int>, duration: <int>, review: <string>,update: <bool> }
Route Handler: routes.post_review
Return Type: JSON Object
Return Parameters: { message (string) }

Route: /review/:geo_type/:geo_id/:user
Description: Returns the review for a specific geographic location and user
Request Method: GET
Route Parameter(s):
- **:geo_type** - string, representing the type of geographical location (state, county, metro, or place)
- **:geo_id** - string, representing the identifier of the geographical location
- **:user** - The username of the user who wrote the review

Query Parameter(s): None
Route Handler: routes.get_review
Return Type: JSON Object
Return Parameters: { reviewer (string), geo_type (string), geo_id (string), lived (int), duration(int), stars (int), review (string) }

Route: /review/:geo_type/:geo_id/:user
Description: Deletes a review for a specific geographical location written by a user
Request Method: DELETE
Route Parameter(s):
- **:geo_type** - string, representing the type of geographical location (state, county, metro, or place)
- **:geo_id** - string, representing the identifier of the geographical location
- **:user** - The username of the user who wrote the review

Query Parameter(s): None
Route Handler: routes.delete_review
Return Type: JSON Object

Return Parameters: { message (string) }

Route: /user_reviews/:user
Description: Retrieves all reviews given by a specific user.
Request Method: GET
Route Parameter(s):
:**user** - string, The username of the user whose reviews are requested.
Query Parameter(s): None
Route Handler: routes.user_reviews
Return Type: JSON Array
Return Parameters: [{duration (integer), geo_id (string), geo_type (string), lived (integer), name (string), review (string), reviewer (string), stars (integer) },...]

Route: /favorite
Description: Adds a location to a user's favorites list
Request Method: POST
Route Parameter(s): None
Query Parameter(s): None
Request Body: { favorite: <bool> (true to add, false to delete), user: <user id>, geo_id: <geo_id>, geo_type:<geo_type>, get: <bool> (true to get favorite, false if editing) }
Route Handler: routes.favorite
Return Type: JSON Object
Return Parameters: { message (string) }

Route: /user_favorites/:user
Description: Retrieves all favorites (favorited locations) of a specific user.
Request Method: GET
Route Parameter(s):
:**user** - string, The username of the user whose favorites are requested.
Query Parameter(s): None
Route Handler: routes.user_favorites
Return Type: JSON Array
Return Parameters: [{geo_id (string), geo_type (string), name (string), user (string)},...]

Route: /search
Description: Searches for locations based on type of geographical location and other filters. Also enables the user to sort the results in ascending/descending order for a particular metric.
Request Method: GET
Route Parameter(s): None

Query Parameter(s):

:desc - bool (default: true), Sort order for the results
:geos - string, representing the type of geographical location (state, county, metro, or place)
:page - integer, current page of the results
:pageSize - integer, size of the page
:sort - string, metric on which results should be sorted
:filters - JSON Array, [{field (string), min (float), max(float)},..] , search criteria

Route Handler: routes.search
Return Type: JSON Array
Return Parameters: [{geo_id (string), geo_type (string), name (string), metric_1 (float)...},...]

Route: /geo_trends/:geo_type/:geo_name/:metric1/:metric2/:timeframe1/:timeframe2
Description: Retrieves trend data for two metrics of a geographic entity, over two time frames.
Request Method: GET
Route Parameter(s):

:**geo_type** - string, representing the type of geographical location (state, county, metro, or place)
:**geo_name** - string, representing the name of the geographic location
:metric1 - string, The name of the first metric to retrieve trend data for.
:metric2 - string, The name of the second metric to retrieve trend data for
:timeframe1 - integer, starting time frame(year) to fetch the trend data
:timeframe2 - integer, ending time frame(year) to fetch the trend data

Query Parameter(s): None
Route Handler: routes.geo_trends
Return Type: JSON Array
Return Parameters: [{geo_id (string), year (integer), Metric_1 (float), Metric_2 (float)},...]

Route: /geo_detail/:year/:geo_type/:geo_id
Description: Returns the available details of a geographical location for a specific year
Request Method: GET
Route Parameter(s):

:**year** - integer, representing the year for which the details are requested
:**geo_type** - string, representing the type of geographical location (state, county, metro, or place)

:**geo_id** - string, representing the identifier of the geographical location
Query Parameter(s): None
Route Handler: routes.geo_detail
Return Type: JSON Object
Return Parameters: {geo_id (string), name (string), Metric_1 (float)...Metric_n (float)}

Route: /geo_shapes/:geo_type/:metric/:year
Description: Retrieves the shapes (e.g., polygons) for all geographic entities of a given type, for a given year.
Request Method: GET
Route Parameter(s):
:**geo_type** - string, representing the type of geographical location (state, county, metro, or place)
:**year** - integer, representing the year for which the details are requested
:metric - The name of the metric to be used to color the shapes (e.g., 'population', 'income', etc.).
Query Parameter(s): None
Route Handler: routes.geo_shapes
Return Type: JSON Array
Return Parameters: [{geo_id (string), geo_json (json), metric (string), name (string), type (string), value (integer)},...]

Route: /geo_shape/:geo_id
Description: Retrieves the shape (eg: Polygon) for a specific geographic entity, given its unique identifier.
Request Method: GET
Route Parameter(s):
:geo_id - string, representing the identifier of the geographical location
Query Parameter(s): None
Route Handler: routes.geo_shape
Return Type: JSON Object
Return Parameters: { geo_id (string), type (string), geo_json (json)}

Route: /geo_names
Description: Retrieves the names of all geographic entities.
Request Method: GET
Route Parameter(s): None
Query Parameter(s):
Types: Array with subset of {state (string), county (string), metro (string), place (string)}

Route Handler: routes.geo_names
Return Type: JSON Array
Return Parameters: [{name (string), geo_id (string), type (string)},...]

Route: /geo_name/:geo_type/:geo_id
Description: Retrieves the name of a geographic entity, given its type and unique identifier.
Request Method: GET
Route Parameter(s):
:**geo_type** - string, representing the type of geographical location (state, county, metro, or place)
:**geo_id** - string, representing the identifier of the geographical location
Query Parameter(s): None
Route Handler: routes.geo_name
Return Type: String
Return Parameters: name (string)

Route: /geo_keys/:geo_type
Description: Retrieves the list of all names of geographical location given its type
Request Method: GET
Route Parameter(s):
:**geo_type** - string, representing the type of geographical location (state, county, metro, or place)
Query Parameter(s): None
Route Handler: routes.geo_keys
Return Type: JSON Array
Return Parameters: [{name (string),..]

Exhibit B - Select Queries:

Query Descriptions:

1. Query to pull information (all available metrics) specific to a single geographic location for a given year. This query also aggregates reviews to find and average and joins it to the result
2. This query finds all reviews for a specific user and joins them back to the information about the geographic location

3. This query finds all favorites for a specific user and joins them back to the information about the geographic location
4. This query powers our search page and matches all geographic locations that match the search criteria specified by the user. The result is paginated and sorted. This is just a sample as the number of potential filters and unions is rather large
5. Query to populate our trends chart. It pulls historical data bounded by a year range for 2 metrics for a given location and returns them in sorted order.
6. This query powers our heatmap and searches for geographic locations that overlap with the bounding box of the viewport of the map. It then joins resulting shapes back to the location metadata and joins in the requested metric

Query Text:

1.
```
`with avg_review as (
    SELECT geo_id, AVG(stars) as rating from reviews where geo_id =
    '0400000US01'
    group by geo_id
)
SELECT a.name, b.*, ar.rating FROM
states a
inner join state_stats b
on a.geo_id = b.geo_id
LEFT JOIN avg_review as ar
ON a.geo_id = ar.geo_id
where a.geo_id = '0400000US01' and year=2021;`
```

2.
```
`with urev as (SELECT
    r.*
    from reviews r
    WHERE r.reviewer = '<user_id>')
SELECT l.name, r.* from
urev as r, states as l
where r.geo_id = l.geo_id and r.geo_type = 'state'
UNION
SELECT l.name, r.* from
```

```
   urev as r, counties as l
   where r.geo_id = l.geo_id and r.geo_type = 'county'
   UNION
   SELECT l.name, r.* from
   urev as r, places as l
   where r.geo_id = l.geo_id and r.geo_type = 'place'
   UNION
   SELECT l.name, r.* from
   urev as r, metros as l
   where r.geo_id = l.geo_id and r.geo_type = 'metro';`
```

3. `with ufav as (SELECT
   r.* from favorites r WHERE r.user = '<user_id>')
   SELECT l.name, r.* from
   ufav as r, states as l
   where r.geo_id = l.geo_id and r.geo_type = 'state'
   UNION
   SELECT l.name, r.* from
   ufav as r, counties as l
   where r.geo_id = l.geo_id and r.geo_type = 'county'
   UNION
   SELECT l.name, r.* from
   ufav as r, places as l
   where r.geo_id = l.geo_id and r.geo_type = 'place'
   UNION
   SELECT l.name, r.* from
   ufav as r, metros as l
   where r.geo_id = l.geo_id and r.geo_type = 'metro';`
4. `SELECT * FROM(
   SELECT
   a.geo_id, a.name, 'metro' as geo_type, b.med_inc_information,
   b.median_commute
   FROM
   metros as a
   join
   metro_stats as b
   on a.geo_id = b.geo_id
   where

```
            b.year = 2021 AND
            b.med_inc_information BETWEEN 108929 AND 200000 AND
            b.median_commute BETWEEN 0 AND 21
      UNION
      SELECT
            a.geo_id, a.name, 'place' as geo_type,
      b.med_inc_information,b.median_commute
      FROM
            places as a
      join
            place_stats as b on a.geo_id = b.geo_id
      where
      b.year = 2021 AND
      b.med_inc_information BETWEEN 108929 AND 200000 AND
      b.median_commute BETWEEN 0 AND 21
      ) as res
      ORDER BY med_inc_information DESC, name
      LIMIT 50 OFFSET 0;`
```

5. `SELECT b.geo_id, b.year, b.total_population, b.median_commute
   FROM states a
   JOIN state_stats b
   on a.geo_id = b.geo_id
   WHERE a.name LIKE 'utah' and b.year >= 2009
   and b.year <= 2021 ORDER BY b.year;`

6. `with matched as (
      select
      geo_id, geo_json, type
      from
      geo_indexed
      where
      mbrintersects(
       ST_GeomFromText('LINESTRING(-1.6731289332134573
-160.13671875000003,52.37349607662163 -33.57421875000001)',4326),
     geo_index

```
 )
   AND
   type = 'state'
 )
SELECT
geo.geo_id, geo.geo_json, geo.type,
meta.name,
'total_population' as metric,
metrics.total_population as value
FROM
matched geo
INNER JOIN
states meta
ON geo.geo_id = meta.geo_id
INNER JOIN
state_stats metrics
ON geo.geo_id = metrics.geo_id
AND metrics.year = 2021
ORDER BY metrics.total_population desc;`
```

ATTACHMENT 1 : QUERY OPTIMIZATION TABLE

Original Query	Optimized Query	Opimization Technique used
app.get('/geo_detail/:year/:geo_type/:geo_id', routes.geo_detail);	**Pages: DetailsPage.js, ComparisonPage.js**	
Query running time: ~160-170ms for Details Page (used Texas and California geographies)	Optimized running time: ~120-130ms	
SELECT a.name, b.*, c.rating FROM ${geoTypeMap[req.params.geo_type]} a inner join ${req.params.geo_type}_stats b on a.geo_id = b.geo_id LEFT JOIN reviews c ON a.geo_id = c.geo_id where a.geo_id = '${req.params.geo_id}' and year=${req.params.year} ;	with avg_review as (SELECT geo_id, AVG(stars) as rating from reviews where geo_id = '${req.params.geo_id}' group by geo_id) SELECT a.name, b.*, ar.rating FROM ${geoTypeMap[req.params.geo_type]} a inner join ${req.params.geo_type}_stats b on a.geo_id = b.geo_id LEFT JOIN avg_review as ar ON a.geo_id = ar.geo_id	Using Common Table Expression, we were able to reduce the execution time about 24%

`app.post('/favorite', routes.favorite)`

Query running time: ~55ms

```
const favorite = async function(req, res) {
let b = req.body;
let query;
if (b.get){
query = `
select user from favorites where user = '${b.user}' and geo_type='${b.geo_type}' and geo_id = '${b.geo_id}';
`
connection.query(query, (err, data) => {
if (err || data.length === 0) {
console.log(err);
res.json({});
} else {
res.json({isFavorite: data.length>0});
}
});
}else{
if(b.favorite){
query = `
INSERT INTO favorites (user, geo_type, geo_id)

VALUES ('${b.user}','${b.geo_type}','${b.geo_id}');
`
}else{
query = `
delete from favorites where user = '${b.user}' and geo_type='${b.geo_type}' and geo_id = '${b.geo_id}';
`
}
```

DetailsPage.js

Query running time: ~70ms

```
CREATE INDEX userIndex
ON favorites (user);
```

Query part is same after adding the index as the original query

We created a secondary index on user. Since currently there are very few users, we actually saw query run time to go up a little. However, this is expected due to overhead of adding an index on parse data for user. We think this would still be a good optimization as the most commercial sites tend to lot of users.

`app.get('/geo_shapes/:geo_type/:metric/:year', routes.geo_shapes);`	**HeatMap.js**	
Query running time: ~3s (when fetching a zoomed view of places) SELECT geo.*, meta.name, '${req.params.metric}' as metric, metrics.${req.params.metric} as value FROM	Query running time: ~1.5s (when fetching a zoomed view of places) let matched = req.query.coords ? `with matched as (select geo_id, geo_json, type from geo_indexed where	Initially we were fetching all geospatial
`app.get('/geo_trends/:geo_type/:geo_name/:metric1/:metric2/:timeframe1/:timeframe2', routes`	**TrendsPage.js**	
Query running time: ~80ms on metro SELECT b.geo_id, b.year, b.${req.params.metric1}, b.${req.params.metric2} FROM ${req.params.geo_type}_stats b JOIN ${geoTypeMap[req.params.geo_type]} a on a.geo_id = b.geo_id WHERE a.name LIKE '${geo_name}' and b.year >= ${req.params.timeframe1} and b.year <= ${req.params.timeframe2} ORDER BY b.year;	Query running time: ~50-70ms on metro SELECT b.geo_id, b.year, b.${req.params.metric1}, b.${req.params.metric2} FROM ${geoTypeMap[req.params.geo_type]} a JOIN ${req.params.geo_type}_stats b on a.geo_id = b.geo_id WHERE a.name LIKE '${geo_name}' and b.year >= ${req.params.timeframe1} and b.year <= ${req.params.timeframe2} ORDER BY b.year;	We optimized this query by optimizing the join. We used the smaller relation as the outer instead of larger relation to be the outer. This reduced the number of tuples to scan for search and we were able to optimize about 22% performance improvement.

`app.get('/geo_names', routes.geo_names);`

Query running time: ~2 s (1.8 - 2.65 s) on compare page

```
const geo_names = async function(req, res) {
console.log(req.query.types)

let subqueries = req.query.types.map((t)=>{
return (
`SELECT name, geo_id, '${t}' as type from ${geoTypeMap[t]}`
)
})
let query = subqueries.join(' UNION ALL ')
```

ComparisonPage.js

Query running time: ~1.15s

```
const geonamesCache = new NodeCache();
const geo_names = async function(req, res) {
console.log(req.query.types)
if (geonamesCache.has('geo_names')){
console.log("Getting it from cache");

return res.send(geonamesCache.get('geo_names'));
}
else{
let subqueries = req.query.types.map((t)=>{
return (
`SELECT name, geo_id, '${t}' as type from ${geoTypeMap[t]}`
)
})
let query = subqueries.join(' UNION ALL ')
connection.query(
query, (err, data) => {
if (err || data.length === 0) {
console.log(err);
res.json({});
} else {
geonamesCache.set('geo_names', data);
console.log("Getting it from API cache")
res.json(data);
}
});
}
}
```

Since the information to be fetched is static, we used in memory /server side caching using node-cache. which reduced the execution time by 37%

`app.post('/search', routes.search)`

Query running time: ~2 sec

```
let jsn = req.body;
let b = jsn.filters;
let desc = jsn.desc ? 'DESC' : ''
let sort = jsn.sort || 'name';
let selection = b.map((f)=>`b.${f.field} BETWEEN ${f.min} AND ${f.max}`);
let projection = b.map((f)=>`,b.${f.field}`).join('');
let selection_joined = selection.join(' AND ')
let query_body = jsn.geos.map((g)=>{
return(
`SELECT a.geo_id, a.name, '${g}' as geo_type ${projection}
FROM ${geoTypeMap[g]} as a join
${g}_stats as b on a.geo_id = b.geo_id
where b.year = 2021 AND ${selection_joined}`
)
}).join(' UNION ')
let query = `
SELECT * FROM(
${query_body}
) as res
ORDER BY ${sort} ${desc}
LIMIT 100;
`;
```

SearchPage.js

Query running time: ~400 ms

```
let jsn = req.body;
let pageSize = jsn.pageSize || 100;
let page = jsn.page - 1 || 0;
let offset = pageSize * page;
let b = jsn.filters;
let desc = jsn.desc ? 'DESC' : ''
let sort = jsn.sort || 'name';
let selection = b.map((f)=>`b.${f.field} BETWEEN ${f.min} AND ${f.max}`);
let projection = b.map((f)=>`,b.${f.field}`).join('');
let selection_joined = selection.join(' AND ')
let query_body = jsn.geos.map((g)=>{
return(
`SELECT a.geo_id, a.name, '${g}' as geo_type ${projection}
FROM ${geoTypeMap[g]} as a join
${g}_stats as b on a.geo_id = b.geo_id
where b.year = 2021 AND ${selection_joined}`
)
}).join(' UNION ')
let query = `
SELECT * FROM(
${query_body}
) as res
ORDER BY ${sort} ${desc}, name
LIMIT ${pageSize} OFFSET ${offset};
`;

console.log(query)
connection.query(query, (err, data) => {
if (err) {
console.log(err);
res.json([]);
} else {
res.json(data);
}
});
}
```

We used Pagination for displaying the search results; Without pagination, it takes 1.2 seconds to load the results. With Pagination, it only takes 330 ms to load the results resulting in 70% faster perfomance.

app.get('/user_reviews/:user', routes.user_reviews);

Query running time: ~50 ms

```
with urev as (SELECT
r.*
from reviews r
WHERE r.reviewer = '${req.params.user}')

SELECT l.name, r.* from
urev as r, states as l
where r.geo_id = l.geo_id and r.geo_type = 'state'

UNION

SELECT l.name, r.* from
urev as r, counties as l
where r.geo_id = l.geo_id and r.geo_type = 'county'

UNION

SELECT l.name, r.* from
urev as r, places as l
where r.geo_id = l.geo_id and r.geo_type = 'place'

UNION

SELECT l.name, r.* from
urev as r, metros as l
where r.geo_id = l.geo_id and r.geo_type = 'metro'
`, (err, data) => {
if (err) {
console.log(err);
res.json([]);
} else {
res.json(data);
}
});
}
```

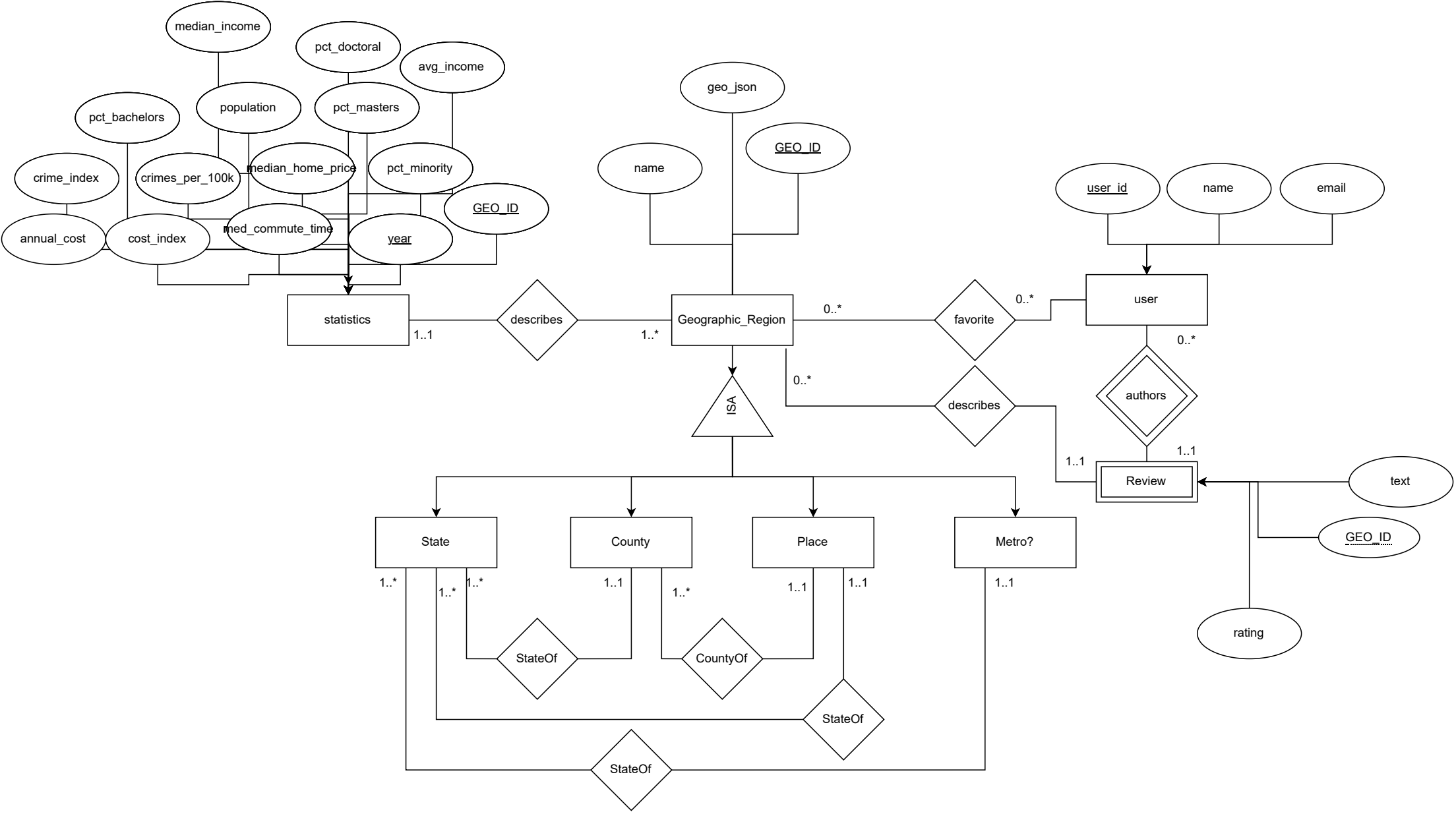
DetailsPage.js

Query running time: ~35 ms

CREATE INDEX userIndex
ON favorites (user);

Remainder part of the query to be the same as the original query

By creating an index on user, we were able to bring the execution time doen by roughly 30% . As with the favorite query, we expect this optimization to scale up with number of users.


median_income
pct_doctoral
avg_income
pct_bachelors
population
pct_masters
crime_index
crimes_per_100k
median_home_price
pct_minority
annual_cost
cost_index
med_commute_time
year
GEO_ID
statistics
1..1
describes
1..*
Geographic_Region
geo_json
name
GEO_ID
0..*
favorite
0..*
user
user_id
name
email
0..*
authors
1..1
Review
text
GEO_ID
rating
0..*
describes
1..1
ISA
State
County
Place
Metro?
1..*
1..*
1..*
1..1
1..*
1..1
1..1
1..1
StateOf
CountyOf
StateOf
StateOf