Smart Travel Planner: Weather, Crowds, and Trips Made Simple

Project Summary

Our project is a web application for travel planning that combines data from various sources into a single, centralized platform. Users can enter their destination and dates, and the system will provide a personalized snapshot of what their trip might look like, including weather, crowdedness, flight options, and popular attractions. Through this project, we aim to give travelers a more intelligent, data-driven approach to trip planning, all while dramatically reducing the time they need to spend planning.

We want this platform to feel like a one-stop shop for trip research. A user can ask, for example, "What's the best time to visit Barcelona in March?" and instantly see weather forecasts, expected crowdedness, suggested flights, and recommended attractions. By combining all this information in a structured format, our system provides travelers with both convenience and confidence in their travel experience.

Application Description

The main issue with today's travel planning is that it's scattered and often overwhelming. Right now, someone might need to use a weather site, a booking site for flights, and even blogs or guides to figure out where to visit. This not only takes a lot of time, but also can result in incomplete information. Our application solves this by pulling together environmental data, flight information, and tourist attraction data into one place.

At its core, our app will make travel planning both simpler and smarter. Through the use of real datasets and APIs, we can give users all the information they need in one query. For example, if a student wants to know the best time to visit Chicago, they can quickly see which weeks are less crowded, what the weather will look like, and which attractions are worth visiting. The idea is to make trip planning less stressful while still letting users explore options and get inspired about where to go.

Creative Component

The creative component of our project is an integrated visualization tool that pulls data from multiple APIs in real time. Instead of static lists, the system will process user inputs and generate personalized suggestions that combine weather conditions, crowd predictions, and flight options within a given budget. For instance, if a user enters “New York in April,” the app could return interactive graphs of temperature trends, bar charts of attraction crowdedness, and a map of recommended spots to visit.

This requires advanced data integration and smart transformations to merge weather APIs, flight search APIs, and tourism datasets into one system. On top of that, the visual component will involve building interactive dashboards with charts, maps, and comparisons, so users can actually explore trade-offs between options instead of just reading static outputs. This makes the application technically challenging while directly improving user experience.

Usefulness

We believe that our proposed application will be useful since it combines many types of information that are typically needed to be checked on numerous apps and websites. For example, people have to use weather apps for temperature and air quality, map apps for crowdedness, and apps like Yelp for popular places. Our application will save time and effort by putting all the features in place. This gives the user a fast and easy way to check city conditions before they make plans to travel.

The basic functions for our app are searching for a destination to view the current temperature, air quality, and flight options at a given time. And the user can browse through lists of popular places. We also allow users to filter on what type of places they want to visit, such as restaurants, attractions, scenic spots, while providing the crowdedness levels as well.

Some of our advanced features will be generating graphs or charts based on user inputs. We can also combine our datasets to recommend the best time to visit a location. Furthermore, we also want to achieve the feature of providing the user with the route and directions for selected locations in the fastest way possible. Essentially, a built-in map allows the user to determine the locations they want to visit.

Realness

Weather and Temperature Data
-Source: OpenWeatherMap API Weather API - OpenWeatherMap
-Format: JSON
-Size: This API provides hourly and daily data worldwide. This easily exceeds one thousand data size.

Air Quality Data

-Source: OpenAQ API openaq.org

-Format: JSON/CSV

-Size: Easily exceeds one thousand data as it includes air quality measurements globally.

Crowdedness/Popular Times/Popular places

-Source: Google Places API Google Maps Platform Documentation | Places API | Google for Developers livebusyness levels/ Yelp Open Dataset for business places Open Dataset | Yelp Data Licensing, limited, good starting Point.

-Format: JSON/CSV

-Size: The Yelp dataset is limited to only 11 cities, but the Google API should be able to fill in the gap for thousands of businesses or locations' data

Flight Options

-Source: OpenFlights Dataset OpenFlights: Airport and airline data

-Format: .dat files

-Size: This contains over 13000 airports worldwide with 600 airlines and over 60000 flight routes. This should be a comprehensive database to start from.

Possible Source for the shortest path features

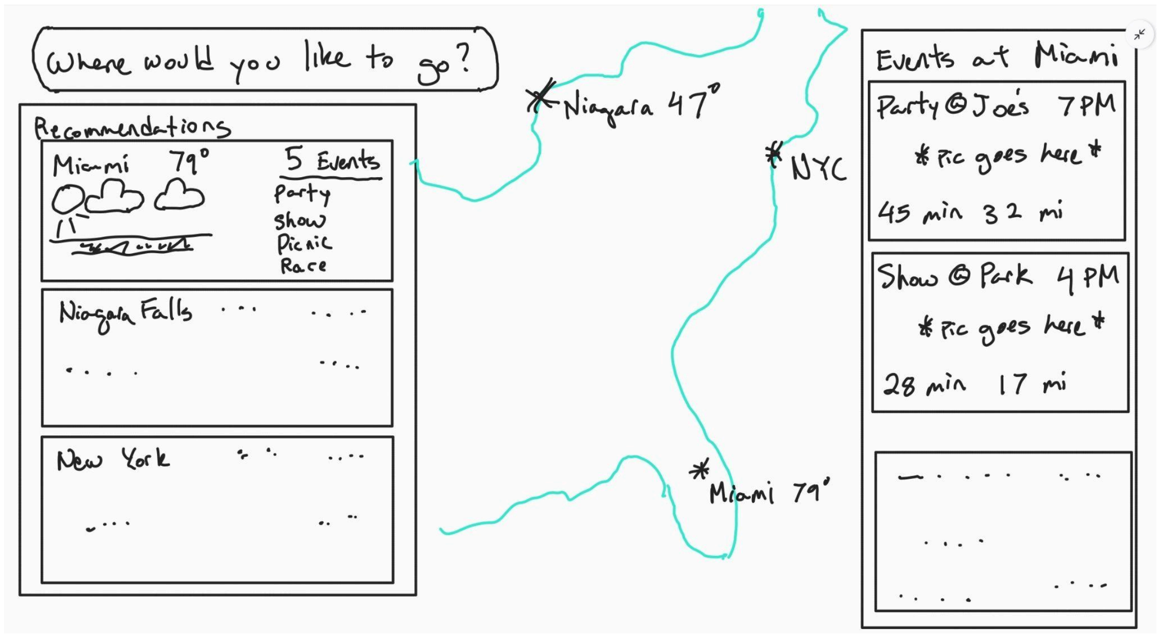
-Sources: Paid Google API/ OpenStreetMap and OpenRouteService for routing OpenStreetMap openrouteservice. The combination of Leaflet and OSM APIs might be able to accomplish a map visualization for the routing feature.

Detailed Description of the Functionality of the Website

1. A clear list of the functionality: Who [add/insert/update/delete] what types of data in your application when? (e.g., the user would submit a form with [what data] when they [want to do something].)
    a. The data will be uploaded to the server via API calls using servers such as Google Maps and OpenWeatherMap. Flight data can be acquired from the OpenFlights Dataset at first and then updated via API calls if necessary.
    b. Trip Query & Search (Flights)
        i. Users will be able to submit a trip query through a search form, with questions such as destination, travel dates, and maybe filters (event types, temperature ranges, and flight costs).
    c. Weather & Environmental Data Integration
        i. When a user submits a destination and date, the system uses APIs to fetch real-time and historical weather data and updates it after some time period when the user will need it again.

    d. Crowdedness & Popularity Insights
        i. The system queries tourism datasets to predict crowd levels at attractions during the requested dates. Popularity data can be brought from sources like Yelp and Google reviews, and maybe from forums and blogs if time permits. This can happen after user queries or when trying to figure out recommended destinations.
    e. Visualization & Dashboards
        i. When a user queries a location, information about the location is displayed using visualization methods such as graphs and lists.

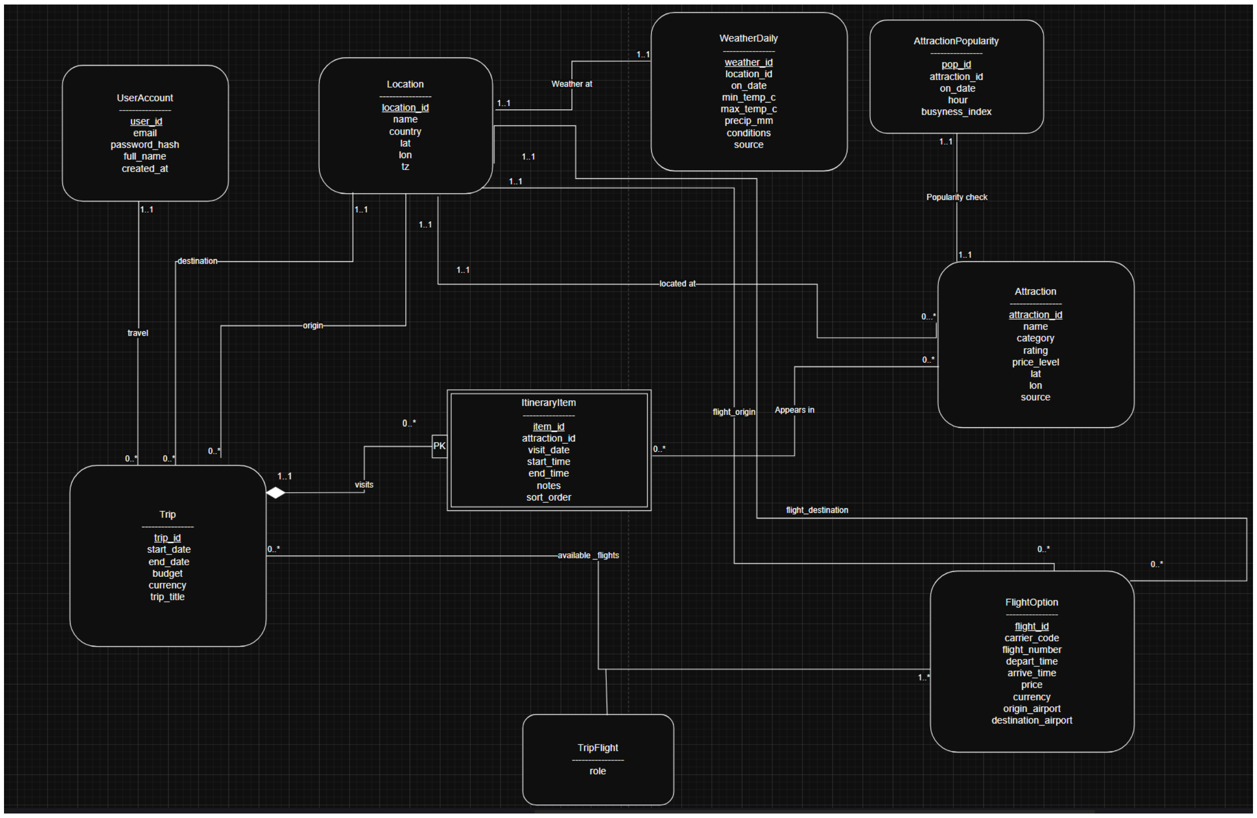
2. A low-fidelity UI mockup: What do you imagine your final application's interface might look like? A PowerPoint slide or a pencil sketch on a piece of paper works!
    a. Home page
    b.



3. Project work distribution: Who will be responsible for each of the tasks or subtasks?
    a. Amogh
        i. Back-end, relational database system, and setting up the database schema to be able to add various API sources to
    b. Kevin
        i. Data mining/acquiring, such as weather, flight. Implementing API calls into the application and calling when necessary.
    c. Orlando
        i. Front-end app, UI, user query interface, Main menu with recommendations, and query result pages.
    d. Preity
        i. Data mining/acquiring, specifically event and attraction data. Implementing API calls into the application and calling when necessary.

4. Explain how backend systems will be distributed across members. Be as specific as possible as this could be part of the final peer evaluation metrics.
    a. Amogh
        i. Build the main trip search API (destination + dates + budget) and database.
        ii. Make calls to the weather, flights, and attractions APIs using the inputted information.
        iii. Process requests and queries to the backend, including request validation and throwing errors.
    b. Kevin
        i. Connect to external weather APIs (forecast and historical).
        ii. Analyse temperature and precipitation trends.
        iii. Build a backend for weather queries by location and date
        iv. Provide queried data formatted for visualization
    c. Orlando
        i. Connect to flight APIs for searching available flights
        ii. Build logic to handle inputs: origin city, destination, dates, budgets
        iii. Implement sorting/filtering.
        iv. Provide queried data formatted for visualization
    d. Preity
        i. Connect to attractions/tourism datasets and APIs.
        ii. Implement crowdedness prediction logic using historical data.
        iii. Build logic to handle inputs: less or more crowded places, cultural or interactive attractions
        iv. Format outputs for visualization (bar charts of crowdedness, maps of attractions).

Stage 2
We are creating a UML Diagram.
UML Diagram:



Explain your assumptions for each entity and relationship in your model. Discuss why you've modeled something as an entity rather than an attribute of another entity. Describe the cardinality of relationships, like why a student is linked to only one advisor. These assumptions might come from customer requirements or application constraints. Please clarify them.

- User Account
    - The user account is an independent entity that represents a user of a program. The user contains identifiable information such as the user_id, email, password, full name, and creation date. It is an independent entity because users can exist without having any trips, and each user can have multiple trips (0..*). Because of the above, it has a 1-to-many relationship with trips, and each trip can only be created by one user.
- Trip
    - The trip is an entity that represents a single user outing. It is modeled as an entity because it is the foundation of a trip, containing the information about that one trip. It is independent of the user because a user can have multiple trips. It contains identifiers such as trip id and user id. Trip descriptions include

origin_location_id, destination_location_id, start_date, end_date, budget, currency, trip_title. A trip must have a user who created it (1..1). It also must have an origin and destination location (1..1), and some flights that connect the two locations (1..*). It can have none or many itinerary items (0..*) as that can be modified easily and on a whim.

- Trip Flight
    - Trip flight is an identity because it is an organization of the Flight Options table specific to any trip. This is useful as it can contain many flights in order to get from the origin location to the destination location. Therefore, it can be connected to any trip (0..*) or no trip at all because it is a suggestion. It can also be connected to multiple flights (1..*) as one flight is needed to get somewhere, but multiple flights might be needed to get to the specific place that you are looking for.
- Flight Options
    - Flight options is an entity because the same flight can be suggested across many users/trips. It also serves as a cache and provides reuse for the options pulled from APIs/datasets. The number of attributes specific to the flight such as the flight id, carrier code, flight number, origin and destination id, depart and arrive times, price, and currency, justifies a standalone table to store API information.
- Location
    - The Location entity represents a geographic area such as a city or region, containing stable information like name, country, latitude, longitude, and time zone. It serves as the foundation for weather data, attractions, and flights. Modeling it as a separate entity rather than as attributes of other tables avoids redundancy and allows multiple datasets, like weather, attractions, and trips, to connect to a single location. Each location can have many weather records and attractions, but is uniquely identified in trip origin and destination relationships.
- Weather Daily
    - WeatherDaily stores daily environmental data for each location, including minimum and maximum temperature, precipitation, and conditions. This information changes daily, so it's modeled as its own entity instead of being part of Location. Each WeatherDaily record is tied to exactly one location, but a single location can have many weather entries over time. The relationship's one-to-many structure reflects that a city's weather evolves across different dates, and storing it separately allows for historical queries and data visualization.
- Itinerary Item
    - The ItineraryItem entity represents an individual event or activity within a user's trip. It connects a trip to a specific attraction while storing extra details such as visit date, start and end time, notes, and sort order. It's modeled as an entity rather than a relationship because it carries meaningful attributes beyond just linking a

trip and attraction. Each trip can include many itinerary items, but every itinerary item belongs to exactly one trip and references a single attraction. This design allows for flexible and detailed scheduling within a trip.

- Attraction
  - The Attraction entity defines points of interest such as landmarks, museums, or restaurants, with attributes like name, category, rating, and price level. It's linked to one Location since each attraction is tied to a specific geographic area. Making attractions a standalone entity allows them to be shared across multiple users and trips, making it possible to aggregate data like popularity trends or recommendations. The one-to-many relationship from Location to Attraction captures that a city typically has many attractions.
- Attraction Popularity
  - AttractionPopularity captures the crowd or busyness level for a particular attraction at a given hour and date. It changes frequently and represents time-based data, which is why it's modeled as its own entity rather than an attribute of Attraction. Each record measures how busy an attraction is during a specific time, and each attraction can have many popularity readings. The one-to-many relationship allows the system to store a full timeline of popularity metrics for crowd prediction and recommendations on when to visit.

Relational Schema

1. **Location**(location_id:INT [PK], name:VARCHAR(160), country:VARCHAR(80), lat:DECIMAL(9,6), lon:DECIMAL(9,6), tz:VARCHAR(40))

Destination(trip_id:INT [PK, FK to Trip.trip_id], location_id:INT [FK to Location.location_id])
Origin(trip_id:INT [PK, FK to Trip.trip_id], location_id:INT [FK to Location.location_id])

The destination and origin tables are the 1 to many relationship set between Location and trip.

2. **flight_origin**(flight_id:INT [PK, FK to FlightOption.flight_id], location_id:INT [FK to Location.location_id])
3. **flight_destination**(flight_id:INT [PK, FK to FlightOption.flight_id], location_id:INT [FK to Location.location_id])

We do the same thing with flight destination and origin since these are also 1 to many relations.

4. **WeatherDaily**(weather_id:INT [PK], location_id:INT [FK to Location.location_id], on_date:DATE, min_temp_c:DECIMAL(5,2), max_temp_c:DECIMAL(5,2), precip_mm:DECIMAL(6,2), conditions:VARCHAR(64), source:VARCHAR(40))

Attraction(attraction_id:INT [PK], location_id:INT [FK to Location.location_id], name:VARCHAR(200), category:VARCHAR(40), rating:DECIMAL(2,1), price_level:TINYINT, lat:DECIMAL(9,6), lon:DECIMAL(9,6), source:VARCHAR(40))

AttractionPopularity(pop_id:INT [PK], attraction_id:INT [FK to Attraction.attraction_id], on_date:DATE, hour:TINYINT, busyness_index:TINYINT)

ItineraryItem(item_id:INT, trip_id:INT [FK to Trip.trip_id], attraction_id:INT [FK to Attraction.attraction_id], visit_date:DATE, start_time:TIME, end_time:TIME, notes:VARCHAR(400), sort_order:INT, (item_id,trip_id)[PK])
For the itinerary item, we have item_id and trip_id together as the primary key b/c the itinerary item is not identifiable without trip_id. It is a weak entity of Trip.

UserAccount(user_id:INT [PK], email: VARCHAR(255) [UNIQUE], password_hash: VARCHAR(255), full_name:VARCHAR(120), created_at: TIMESTAMP)

travel(trip_id:INT[PK] , user_id:INT)

Trip(trip_id: INT [PK] , start_date: DATE , end_date: DATE , budget:DECIMAL(10,2) , currency:VARCHAR(3) , trip_title:VARCHAR(120))

5. **FlightOption** (flight_id : INT [PK] , carrier_code : VARCHAR(12) , flight_number: VARCHAR(12) , depart_time : TIMESTAMP , arrive_time: TIMESTAMP , price : DECIMAL(10,2) , currency: VARCHAR(3))

TripFlight(trip_id:INT [FK to Trip.trip_id], flight_id : INT [FK to FlightOption.flight_id] ,role: VARCHAR(16), (trip_id , flight_id)[PK])

BCNF Check

- UserAccount –
  - Every non-key attribute (email, password_hash, full_name, created_at) depends only on user_id, the primary key.
- Location –
  - Attributes (name, country, lat, lon, tz) depend solely on location_id.
- Trip –

    - Each trip is identified by trip_id; all other attributes (start_date, end_date, budget, currency, trip_title) depend entirely on that key.
  - Origin / Destination –
    - Each table's key (trip_id) uniquely determines the associated location_id.
  - FlightOption –
    - flight_id uniquely determines all flight details (carrier_code, flight_number, depart_time, etc.).
  - flight_origin / flight_destination –
    - Each flight_id determines exactly one location_id.
  - TripFlight –
    - Composite key (trip_id, flight_id) uniquely identifies each record and determines the role.
  - WeatherDaily –
    - weather_id is the primary key, and all attributes (location_id, on_date, min_temp_c, etc.) depend only on it.
  - Attraction –
    - All descriptive attributes depend solely on attraction_id.
  - AttractionPopularity –
    - pop_id uniquely identifies each record; attraction_id, on_date, and hour are dependent on that key only.
  - ItineraryItem –
    - The composite key (item_id, trip_id) determines all other attributes (attraction_id, visit_date, start_time, etc.).
  - travel –
    - trip_id is both the primary key and a foreign key to Trip.trip_id; it determines exactly one user_id.