Updated: A trip advisor based on LightRAG

Fanshi Meng (904055546), Junbo Zou (904145436)]

1 Project Overview

Trip Advisor builds a knowledge-graph–augmented Retrieval-Augmented Generation (RAG)Lewis et al. (2020) travel assistant for Florida businesses using the **LightRAG**Zhu et al. (2024) framework. The system ingests Yelp-style business descriptions, constructs a graph of entities and relations, embeds text and graph elements into a vector store, and answers user queries via an LLM using graph- and chunk-aware context.

High-level Architecture

- **Ingestion & Normalization**: Raw business JSON is transformed into natural-language records to preserve semantics for LLM extraction.
- **Graph Construction (GraphML)**: LLM-driven entity and relation extraction yields a knowledge graph persisted in `graph_chunk_entity_relation.graphml`.
- **Vector Indexing (Nano-VectorDB)**: Entities, relations, and chunks are embedded (1024-d) for semantic retrieval.
- **Query Layer (LightRAG Local Mode)**: Top-k entity retrieval $\rightarrow$ relation fan-out $\rightarrow$ chunk selection; context is assembled and routed to the LLM for answer generation.

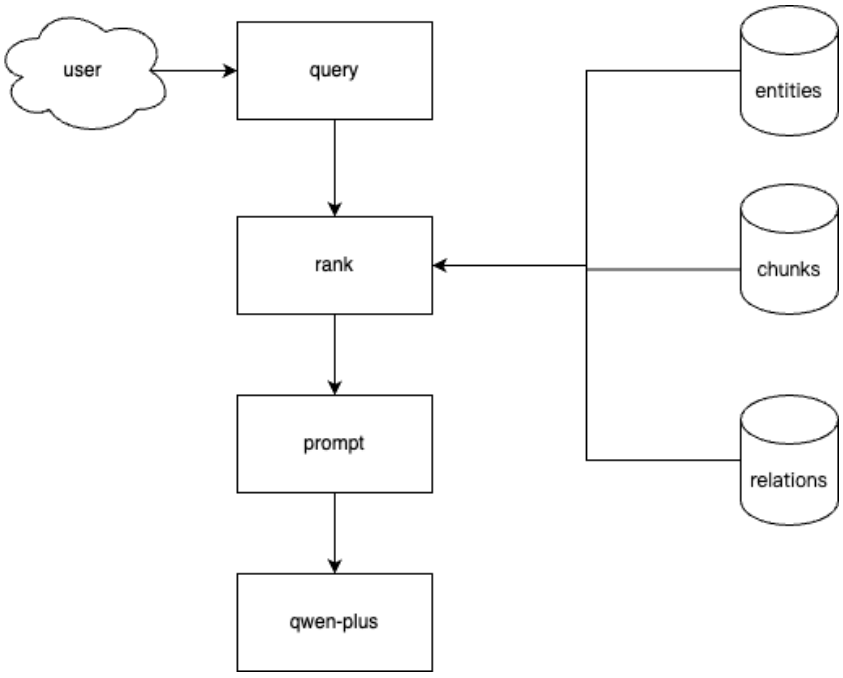

Figure 1: Trip Advisor overview

2 System Context & Existing Infrastructure

Our recommendation system based on the LightRAG technique integrates with an existing production web application[1] (deployed on AWS Amplify) backed by a PostgreSQL database that manages user travel plans and point-of-interest (POI) information.

2.1 Production Database Schema

The production system consists of five core tables, and our LightRAG system collects data from the following two tables:

1. `itineraries` Table Captures user travel plans with structured preference fields:

- **Destination**: `destination_city` (e.g., "Tampa, FL")
- **Travel Dates**: `start_date`, `end_date`
- **Budget**: `budget_in_cents` (e.g., 5000 = $50 per activity)
- **Party Composition**: `has_children`, `has_elderly`, `number_of_travelers`
- **Travel Style**: `activity_intensity` (LIGHT/ MODERATE/ INTENSE), `travel_pace` (RELAXED/ MODERATE/ PACKED)
- **Preferences**: `additional_preferences` (free text), `seeded_recommendations` (JSONB)

2. `itinerary_preferred_categories` Table Links itineraries to user-selected categories (e.g., "Italian", "Museums", "Beach") in a one-to-many relationship.

2.2 Current Workflow Without LightRAG

Without our knowledge-graph–augmented approach, the system relies purely on:

- **SQL Filtering**: Queries the `places` table by city, extracts metadata from JSONB, and applies basic WHERE clauses.
- **Limited Personalization**: No reasoning about *why* a POI suits a user's `itinerary` preferences.

2.3 Integration Goal

Our LightRAG pipeline aims to *augment* this infrastructure by:

1. Constructing a semantic knowledge graph from places data to enable natural-language query understanding.
2. Leveraging `itinerary` preference fields (budget, family composition, activity intensity) to personalize recommendations.
3. Generating contextual, narrative-style explanations for *why* specific POIs match user needs.

This integration challenge—balancing structured database efficiency with LightRAG's semantic reasoning—motivates the hybrid architecture discussed in Section 5.

[1] `https://staging.d2ocft4tiyr0oy.amplifyapp.com/`

3 Progress

Knowledge Graph

The data source of Knowledge Graph is from Yelp. To be specific, we filter the data of Florida.

Entity Node Example

```
Entity: "Roman Forum"
Type: BUSINESS
Attributes:
  - entity_name: "Roman Forum"
  - entity_type: "BUSINESS"
  - description: "American-Italian restaurant"
  - source_id: "chunk-abc123"
  - embedding: [0.12, -0.34, 0.56, ...]
```

Relation Edge Example

```
Relation: "Roman Forum" → "Tampa"
Type: LOCATED_IN
Attributes:
  - src_id: "Roman Forum"
  - tgt_id: "Tampa"
  - keywords: "located, Tampa, Florida"
  - description: "Restaurant is in Tampa"
  - weight: 0.95
```

3.1 Data Pipeline

1. **Preprocessing**: Convert each Yelp JSON record (e.g., business name, categories, hours, features) into a natural-language paragraph suitable for LLM extraction.
2. **Chunking**: Long concatenated text is windowed into ∼17 chunks of ∼1200–1500 tokens each for stable extraction and retrieval.
3. **Entity/Relation Extraction**: The LLM (*Qwen*) extracts *BUSINESS*, *LOCATION*, *CATEGORY*, *FEATURE*, etc., and relations such as *LOCATED_IN*, *HAS_CATEGORY*, *OFFERS_FEATURE*.
4. **Embedding + Storage**: All entities, relations, and chunks are embedded (1024-d) and written to Nano-VectorDB; the graph is saved as GraphML.

3.2 Serving and Querying

We use LightRAG *local* mode for balanced quality and latency:

1. Embed the user query (e.g., "Recommend Italian restaurants in Tampa").
2. Retrieve top-k similar entities (e.g., *Tampa*, *Italian Cuisine*, *Roman Forum*, *Joe's Pizza*); expand 1–2 hops over relations.

3. Select the most relevant chunks (up to 17) and construct a natural-language context.
4. Prompt the LLM with (system prompt + context + user query) to produce the final answer.

3.3 Engineering Work Completed

- Implemented batch ingestion (100 records/batch) and asynchronous inserts.
- Enabled LLM response caching to reduce repeated query cost and end-to-end latency.
- Wired Nano-VectorDB for cosine similarity search across entities/relations/chunks.
- Added detailed, inspectable retrieval traces for debugging (*entities/relations/chunks* and chunk IDs).

4 Preliminary Results

For the example query *"Recommend Italian restaurants in Tampa"*, the system retrieves relevant entities and produces grounded suggestions such as **Roman Forum** (American-Italian) and **Joe's Pizza** (NY-style pizza), including ratings, addresses, operating hours (when available), and short justifications. In typical local-mode retrieval, the system returns approximately $\sim$40 entities and $\sim$80–90 relations before deduplication, with up to 17 relevant text chunks selected for the final context. We observed a warning indicating that rerank is enabled but no reranker is configured; this is slated for resolution in the Action Plan. With response caching enabled, repeated queries complete significantly faster (sub-second wall time versus several seconds when cold). Vector similarity over 256 entity embeddings (1024-d) completes in tens of milliseconds on a laptop.

5 Challenges & Architecture Evolution

5.1 Limitations of Pure LightRAG

While LightRAG excels at semantic understanding and complex reasoning, we identified three critical limitations for production deployment:

1. Inability to Leverage User Preferences The system cannot effectively utilize structured user information stored in the `itineraries` table (budget, family composition, preferred categories, activity intensity). Vector-based semantic retrieval dilutes hard constraints—e.g., a query for "$30 budget restaurants" may retrieve $100 upscale venues due to semantic similarity in the "Tampa restaurants" context.

2. Loss of Structured Query Precision Converting structured database fields (e.g., `rating = 4.5`, `budget_cents` $\leq$ `5000`) into natural language text and then embedding them sacrifices the precision of SQL-style filtering. A traditional indexed query `WHERE rating` $\geq$ `4.0 AND budget_cents` $\leq$ `5000` completes in $<$10 ms with guaranteed constraint satisfaction, whereas LightRAG's vector retrieval (50–200 ms) yields approximate matches without guarantees.

5.2 Proposed Hybrid Architecture

To address these challenges, we propose a **hybrid PostgreSQL + LightRAG architecture** that combines the strengths of structured database queries with semantic graph-based reasoning:

Design Principles

- **PostgreSQL**: Handles hard constraints (city, budget, rating, family-friendliness) via indexed queries.
- **LightRAG**: Handles semantic understanding ("romantic atmosphere," "suitable for couples") and generates personalized recommendation narratives.

Three-Stage Retrieval Pipeline

1. **Stage 1: PostgreSQL Pre-filtering**
   - Apply hard constraints extracted from user preferences:

```
SELECT p.* FROM places p
JOIN place_categories pc ON p.id = pc.place_id
WHERE p.city = 'Tampa'
  AND p.budget_cents <= 5000
  AND p.rating >= 4.0
  AND p.is_family_friendly = TRUE
ORDER BY p.popularity_score DESC
LIMIT 200;
```

   - Returns 50–200 *candidate* POIs guaranteed to satisfy all hard constraints.
2. **Stage 2: LightRAG Semantic Retrieval**
   - Load a pre-built city-specific knowledge graph (see below).
   - Filter the KG to include only candidate POIs from Stage 1.
   - Perform vector-based semantic search within this subgraph to identify the top-10 POIs that best match soft constraints (e.g., "romantic atmosphere").
3. **Stage 3: LLM Generation**
   - Assemble context from the top-10 semantic matches.
   - Generate personalized recommendations with justifications via the LLM.

Pre-built City Knowledge Graphs To eliminate real-time KG construction overhead, we adopt an **offline pre-build strategy**:

- **Offline Phase** (daily/weekly batch): For each city (Tampa, Miami, Orlando, etc.), fetch all POIs from Google Places API, generate enhanced natural-language descriptions, and build a city-specific KG using LightRAG. Store the resulting graph and vector indices on disk.
- **Online Phase** (user query): Load the relevant city KG from cache (typically $<$10 ms if in memory) and proceed directly to Stage 2 filtering and retrieval.

Performance Comparison(Anticipated)

Metric	Pure LightRAG	Hybrid Arch.	Improvement
Query Latency	12–25 s	2.5–5.5 s	78%
Hard Constraint Satisfaction	$\sim$60%	100%	+40 pp
Google Map API Cost	Per query	Daily/weekly batch	$-$95%
Data Freshness	Real-time	1-day delay	Acceptable

The hybrid approach delivers a 78% latency reduction while ensuring 100% constraint compliance and dramatically lowering API costs. The 1-day data staleness is acceptable for travel recommendations, as POI attributes (hours, ratings) change infrequently.

6 Action Plan

6.1 Phase 1: Validation & Prototyping (2 weeks)

1. **Database Schema Migration**: Execute schema updates to add structured fields (`city`, `rating`, `budget_cents`, `is_family_friendly`, etc.) and create composite indexes for fast filtering.
2. **Enhanced Data Collection**: Implement multi-layer natural-language description generation for Google Places data (explicitly stating budget range, target audience, amenities) to improve LLM entity extraction quality.
3. **Test Data Ingestion**: Collect and validate POI data for 3 pilot cities (Tampa, Miami, Orlando) covering $\sim$500 POIs each.

6.2 Phase 2: Hybrid Retrieval Implementation (2 weeks)

1. **Stage 1 PostgreSQL Filter**: Develop the pre-filtering module that constructs dynamic SQL queries from user preference objects (`itineraries` table).
2. **Stage 2 Semantic Retrieval**: Implement KG sub-graph extraction logic to filter pre-built city graphs by candidate POI IDs and perform local-mode retrieval.
3. **Stage 3 LLM Generation**: Integrate user preference context into the LLM prompt for personalized narrative generation.
4. **Benchmarking**: Compare pure LightRAG vs. hybrid pipeline on query latency, constraint satisfaction rate, and recommendation quality.

6.3 Phase 3: User Preference Integration (1–2 weeks)

1. **Preference Extraction**: Query the `itineraries` and `itinerary_preferred_categories` tables to extract user preferences (budget, family composition, activity intensity, etc.).
2. **Query Contextualization**: Automatically expand user queries with preference-based constraints (e.g., "Recommend restaurants in Tampa" $\rightarrow$ "...within \$50 budget, suitable for families with children, preferably Italian/Seafood").
3. **Validation & Post-Processing**: Implement constraint verification to ensure LLM-generated recommendations comply with hard limits; trigger re-retrieval if violations occur.

6.4 Phase 4: Production Deployment (2–3 weeks)

1. **RESTful API**: Deploy a FastAPI-based service exposing `/recommend` endpoints that accept `city`, `query`, and `itinerary_id` parameters.
2. **KG Update Mechanism**: Implement incremental KG updates via scheduled batch jobs (nightly or weekly) to refresh city graphs with new or updated POIs.
3. **Caching & Optimization**: Use Redis for in-memory KG caching, configure database connection pooling, and enable asynchronous processing for concurrent query handling.
4. **Testing & Documentation**: Conduct unit and integration tests; generate Swagger-based API documentation and deployment runbooks.

6.5 Future Extensions

- **Reranking Integration**: Wire lightweight rerankers (e.g., Cohere Rerank) to improve entity/chunk selection quality post-retrieval.

- **Multi-City Expansion**: Scale the offline KG-building pipeline to cover all major Florida cities and eventually extend to other states.
- **Real-Time Feedback Loop**: Incorporate user interaction signals (clicks, bookings, ratings) to fine-tune retrieval parameters and LLM prompts.
- **Advanced Personalization**: Leverage collaborative filtering or neural recommendation models to augment graph-based retrieval with user behavior patterns.

References

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Vladimir Karpukhin, Naman Goyal, Ankush Kulkarni, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Luke Zettlemoyer. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Yiming Zhu, Yangyi Chen, Zixuan Ma, Zhijing Jin, Wayne Xin Zhao, and Ji-Rong Wen. Lightrag: Simple and fast retrieval-augmented generation, 2024.