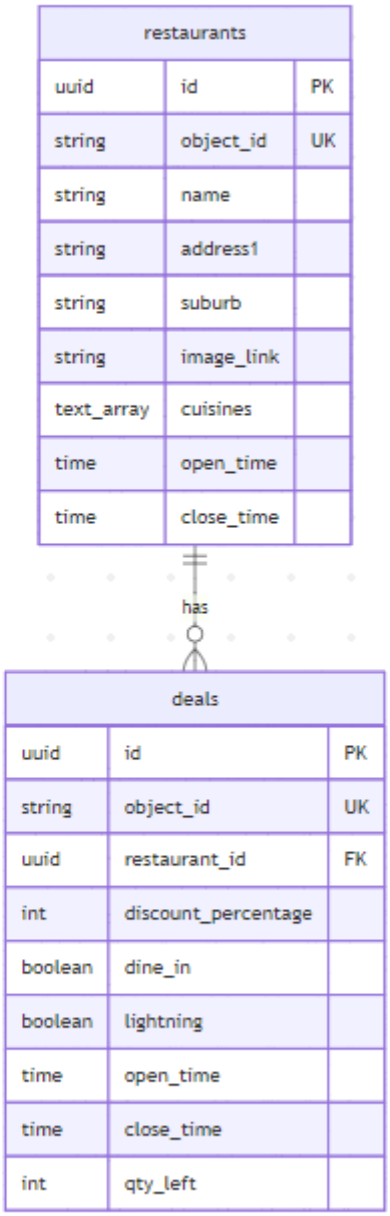
Database Schema Design

Two tables focused on Task 1 (active deals) and Task 2 (peak time). Simple for current requirements, clear evolution path for scaling.

Database Choice: PostgreSQL

I went with PostgreSQL because time-based queries are at the core of what we're doing - filtering deals by time of day and calculating peak windows. PostgreSQL's native TIME type makes range queries efficient, and it handles the complex overlapping time window calculations for peak time well. The data is clearly relational (restaurants have deals), so a relational DB makes sense. Could have used MySQL, but PostgreSQL's window functions and better time handling made it the better fit.

Schema Design



Key Design Decisions

1. Deal Time Inheritance

What: Deals can have their own hours or inherit from restaurant hours **Why:** Looking at the data, some deals specify `open: "3:00pm", close: "9:00pm"` while others don't have times at all **Implementation:** Made deal times nullable, use `COALESCE(deal.open_time, restaurant.open_time)` in queries **Trade-off:** Adds query complexity but matches the business reality

2. TIME Type for Temporal Queries

What: Store times as PostgreSQL TIME instead of strings **Why:** Both APIs filter by time of day - need efficient range queries **Benefit:** Parse once during sync, not on every query. Database validates time values **Limitation:**

TIME doesn't handle midnight-crossing (11pm-2am). Current data is safe (all close before midnight), but late-night restaurants would break this

3. Cuisines as Array

What: Store cuisines as PostgreSQL `TEXT[]` array instead of normalized tables **Why:** The API provides cuisines array, but we don't filter or return them in `/deals` or `/peak-time` **Benefit:** Simpler schema (2 tables instead of 4), no joins for unused data **Future-proof:** If we add cuisine search later, can still query with `WHERE 'Indian' = ANY(cuisines)` or normalize then

4. Proper Data Types

What: Convert API strings to database types **Examples:** `"true"` → BOOLEAN, `"50"` → INTEGER, `["Indian", "Thai"]` → TEXT[] **Why:** Database-level validation, cleaner queries, no parsing overhead

5. UUID Primary Keys

What: UUID instead of auto-increment integers **Why:** Matches external API format, better for distributed systems

---

Schema Summary

2 tables:

- `restaurants` - core restaurant data including cuisines array
- `deals` - one-to-many with restaurants

Why simpler than normalized cuisines?

- `/deal` and `/peak-time` don't filter by cuisine or return cuisine data
- PostgreSQL array support is efficient for storing and querying when needed
- Can normalize later if cuisine search becomes a major feature
- Keeps design focused on current requirements

Trade-offs and Limitations

No Historical Data - Can't track deal performance over time. Fine for current requirements (active deals only). Add versioning if analytics needed.

Peak Time Calculation is Expensive - Full table scan on every request. Small dataset makes it acceptable. Cache with 15-30min TTL if traffic spikes.

Validation in App Layer - No CHECK constraints in DB. Single app is fine. Risk if multiple services write to DB.

Scaling for EatClub Growth

Read Replicas First - Workload is read-heavy (`/deals` queries vs occasional writes). Route GET `/deals` to replicas, writes to primary. Challenge: Lightning deals with `qty_left` decrementing - replication lag (50-

200ms) means users see sold-out deals. Acceptable for most use cases.

Caching Before Sharding - Cache peak time (15-30min TTL) and popular deals. Simpler and cheaper than distributed complexity.

Sharding (Last Resort) - Shard by `restaurant_id` to keep joins together. `/deals` works well. `/peak-time` (peak time) is the problem - needs ALL deals across shards. Either scatter-gather or move to analytics DB. Only worth it if replicas + caching can't handle load.

Consideration: Vertical scaling (bigger instance) + read replicas + caching may handle most growth for a deals platform.