

* The data is highly structured with clear relationships (restaurants, deals, cuisines).
* Relational DB PostgreSQL supports array types (useful for cuisines), JSON fields (for flexible extensions), and advanced indexing for performance.
* It is scalable, widely adopted, and integrates well with modern API frameworks

Thought process behind ER Diagram

1. **Central Entities:** The diagram centres around Restaurants and Deals as the primary entities, indicating a core focus on linking specific deals to specific restaurants.
2. **Denormalization/Cross-Reference:** The Deals\_at\_Restaurant (Cross Reference) table suggests a many-to-many relationship between Restaurants and Deals, and potentially a decision to store Quantity\_available at this intersection, indicating deal stock is specific to a restaurant which also gets frequently updated when customers use the coupon.
3. **Restaurant Timings:** Restaurant\_Timings is separated from Restaurants, implying a restaurant can have multiple timing rules (e.g., for different days or special events/holidays via Effective\_Date).
4. **Operational Details:** Restaurant\_Operations seems to capture operational aspects like Cuisine\_Type and Image\_Link, possibly to keep the main Restaurants table cleaner or to support multiple operational profiles.
5. **Simplified Deal Attributes:** The Deals entity itself appears focused on core deal attributes like discount and flags (Dine\_In\_Flag, Is\_Lightning\_Flag), suggesting that deal availability and quantity are handled at the Deals\_at\_Restaurant level.

**Table Update Rationale**

1. Restaurants: Updated when new venues are added or details change (address, name, etc.).
2. Restaurant\_Operations: Updated for changes in cuisine type or branding/image.
3. Restaurant\_Timings: Updated for schedule changes, holidays, or closures.
4. Deals: Updated when deal terms or flags change.
5. Deals\_at\_Restaurant: Updated for deal assignments to venues and inventory changes (e.g., deal redemption).

**Performance and Scalability with Volume Growth**

1. Implementing indexes on key columns within the tables will significantly improve query performance.
2. Utilizing materialized views where appropriate can help precompute and store complex query results, reducing the load on the database during runtime.
3. Combining materialized views with effective caching strategies when serving API requests can further enhance response times and scalability.
4. Smart API design can leverage pagination, filters etc to provide manageable, efficient responses for web and mobile applications, enhancing both performance and user experience.