Below is an example answer that you could use in your interview to explain the system design, database normalization, and architectural options for the widget/inventory system.

1. What Does This System Do?

At its core, this system appears to be an inventory and pricing management solution for a business that sells various “traps” (the “widgets”) in different packaging sizes to different customer channels. Key functions likely include:

- **Product Catalog Management:** Managing different trap types (e.g., Ant Trap, Mouse Trap, Bear Trap, etc.) with attributes such as cost and supplier information.
- **Customer-Specific Pricing & Packaging Options:** Offering different packaging options and price points based on the customer (e.g., Home Place, Bug Store, No Bears R Us). This could indicate that prices and even available packaging might be tailored to different retail channels.
- **Inventory and Warehouse Management:** Tracking quantities and minimum stock levels (min_qty) across different warehouses (e.g., AUS, ATL, MSP) to ensure adequate inventory and trigger reorder processes.
- **Supplier Integration:** Linking each product with a supplier (e.g., Little Traps, Big Traps, Raytheon) and recording associated costs.

2. Proposed Normalized Database Schema

A normalized schema helps reduce redundancy and improve data integrity. One way to design the schema is to break down the information into discrete entities. Below is one approach with key tables and relationships:

A. Core Entities

Product (Widget)

- **Attributes:**
  - *ProductID* (Primary Key)
  - *Name* (e.g., “Ant Trap”, “Mouse Trap”, etc.)
  - *Cost* (e.g., $0.50, $1, etc.)
  - *SupplierID* (Foreign Key referencing Supplier)

Supplier

- **Attributes:**
  - *SupplierID* (Primary Key)

- *SupplierName* (e.g., “Little Traps”, “Big Traps”, “Raytheon”)

Packaging

- **Attributes:**
  - *PackagingID* (Primary Key)
  - *Description* (e.g., “bag of 10”, “box of 1”, “crate of 1”)

Customer

- **Attributes:**
  - *CustomerID* (Primary Key)
  - *CustomerName* (e.g., “Home Place”, “Bug Store”, “No Bears R Us”)

Warehouse

- **Attributes:**
  - *WarehouseID* (Primary Key)
  - *LocationCode* (e.g., “AUS”, “ATL”, “MSP”)

B. Relationship/Transactional Entities

Pricing / Product Offering

- **Purpose:** Captures the price for a given combination of product, packaging, and customer.
- **Attributes:**
  - *PricingID* (Primary Key)
  - *ProductID* (Foreign Key)
  - *PackagingID* (Foreign Key)
  - *CustomerID* (Foreign Key)
  - *Price* (e.g., $9, $5, $10, etc.)

Inventory

- **Purpose:** Tracks inventory levels for each product at a given warehouse.
- **Attributes:**
  - *InventoryID* (Primary Key)
  - *ProductID* (Foreign Key)
  - *WarehouseID* (Foreign Key)
  - *Quantity* (e.g., 112, 200, etc.)
  - *MinQty* (e.g., 50, 10, etc.)

C. Benefits of This Normalization

- **Data Integrity:** Supplier and cost details reside in the Product table, so if cost updates occur, they're managed in one place.
- **Flexibility:** Pricing is decoupled from the product. This allows multiple packaging options or customer-specific prices without duplicating product information.
- **Scalability:** The separation of inventory by warehouse enables easy tracking and management of stock levels across different locations.

3. Reasonable System Architecture

Here are two approaches you might discuss:

A. Monolithic Application (Simpler Use Case)

- **Presentation Layer:** A web-based UI or dashboard for users to view products, inventory, and pricing information.
- **Business Logic Layer:** A server-side application (e.g., built with Node.js, Python/Django, or Java/Spring) that handles data validation, pricing rules, reorder alerts, etc.
- **Data Layer:** A relational database (e.g., PostgreSQL or MySQL) that implements the normalized schema.
- **Pros:**
    - Faster initial development
    - Simple deployment for small to medium-scale operations

B. Microservices Architecture (Scalable/Enterprise-Level)

- **API Gateway:** Acts as the entry point for client applications, routing requests to appropriate microservices.
- **Separate Microservices:**
    - **Product Service:** Manages product catalog and supplier relationships.
    - **Pricing Service:** Handles customer-specific pricing and packaging configurations.
    - **Inventory Service:** Manages stock levels, warehouse details, and triggers alerts/reorders.
    - **Order/Transaction Service (if needed):** Processes orders and updates inventory.
- **Data Layer:** Each microservice could manage its own database (or use a shared, well-partitioned database), ensuring that each service can scale independently.
- **Pros:**

    - Better scalability and fault isolation
    - Easier to integrate with external systems (e.g., supplier APIs, customer portals)
    - More agile updates to individual components

4. Questions/Concerns to Address Before Finalizing the Design

Before establishing an ideal solution, you might want to clarify the following with the interviewer or stakeholders:

- **Customer-Pricing Relationship:**
    - Is pricing always customer-specific?
    - Can a product be sold to multiple customers with different packaging options and prices simultaneously?
- **Packaging Details:**
    - Are packaging options static, or can they change over time (e.g., seasonal packages or promotions)?
- **Inventory Management:**
    - Are the quantities in the provided dataset representative of the entire inventory per product, or do they vary by packaging/variant?
    - How are inventory updates managed (real-time, periodic batch updates, etc.)?
- **Warehouse Operations:**
    - Are warehouses independent (with their own pricing/inventory rules), or is there a centralized control system?
    - How should inter-warehouse transfers or multi-location orders be handled?
- **Supplier Relationships:**
    - How dynamic is the supplier information?
    - Will products ever have more than one supplier, or is it always a one-to-one relationship?
- **Scalability and Reporting Needs:**
    - Is historical pricing and inventory data needed for analytics?
    - Should the system support auditing and tracking changes over time?
- **Integration and Future Requirements:**
    - Will the system need to integrate with e-commerce platforms, order fulfillment systems, or ERP systems?
    - Are there any performance or scalability requirements that might drive the choice between a monolithic or microservices architecture?

5. Alternative Approaches

Denormalized / Simpler Table Approach

- **When to Consider:** If the dataset is small and the system is not expected to scale significantly, you might opt for a denormalized table that combines all data into one structure. This can simplify queries at the cost of data redundancy and potential anomalies when updating.
- **Trade-offs:**
  - **Pros:** Easier to implement and faster for read-heavy operations on a small dataset.
  - **Cons:** Difficult to maintain consistency as the system grows; changes in supplier or packaging details would need to be updated in multiple rows.

Data Warehousing for Analytics

- **When to Consider:** If historical data and analytics become a requirement (for sales trends, inventory forecasting, etc.), you might design a separate data warehouse that periodically extracts data from the operational database. This could be structured in a star schema for reporting.
- **Trade-offs:**
  - **Pros:** Optimized for read queries and analytical workloads.
  - **Cons:** Increased complexity and potential data synchronization challenges.

Summary

- **System Purpose:** The system is designed to manage an inventory of trap products with customer-specific pricing and packaging, track supplier costs, and maintain stock levels across multiple warehouses.
- **Normalized Schema:** Breaks down into core tables (Product, Supplier, Packaging, Customer, Warehouse) and relationship tables (Pricing, Inventory) to avoid redundancy and improve data consistency.
- **Architecture Options:** Depending on scale and future needs, you could opt for a monolithic application for simpler use cases or a microservices architecture for better scalability and modularity.
- **Open Questions:** Clarifications regarding pricing, packaging, inventory updates, supplier dynamics, and integration requirements are essential to ensure the solution meets all business needs.

This multi-angle approach not only shows your ability to design a normalized database schema but also demonstrates critical thinking about system architecture and potential edge cases.