Database Design Decisions & Tradeoffs

E-Commerce Clothing Store - Database Architecture

Key Architecture Decisions

1. Two Item Tables: `cart_items` + `order_items`

Decision: Separate tables instead of single unified `items` table

Reasoning:

- `cart_items` = temporary, can be deleted with carts (CASCADE)
- `order_items` = permanent historical record, immutable
- Prevents data loss when cleaning up abandoned carts
- Different lifecycles require different deletion policies

Tradeoff:

- ✅Pro: Cart cleanup doesn't affect order history
- ✅Pro: Order items remain even if product deleted
- ❌Con: Must copy data from cart_items → order_items at checkout
- ❌Con: Two tables to maintain instead of one

Implementation: At checkout, copy cart_items to order_items with product snapshot (title, description, image, etc.)

---

2. Product/Product_Variant Separation

Decision: Keep products and variants in separate tables

Reasoning:

- Need to track inventory per size/color combination
- Different variants can have different prices (e.g., XL costs more)
- Each variant needs its own SKU for warehouse fulfillment
- Different colors need different product images

Tradeoff:

- ✅Pro: Proper inventory management per variant
- ✅Pro: Flexible pricing per size/color
- ✅Pro: Clear SKU tracking
- ❌Con: More complex queries (requires JOIN)
- ❌Con: Slightly more setup work for admins

Alternative Considered: Single products table with JSON arrays for sizes/colors

- ❌Rejected: Can't track stock per size, can't price differently

---

3. Denormalization in Orders & Returns

Decision: Store user/address data directly in `orders` and `returns` tables

Reasoning:

- Orders/returns are immutable historical records
- User might change name/email/address later
- Need exact snapshot of what existed at purchase time
- Legal/accounting requirements for exact historical data

Fields Denormalized:

- Orders: `user_email`, `user_first_name`, `user_last_name`, `address_line1`, `address_line2`, `address_city`, `address_zip`, `address_country`
- Returns: `user_email`, `user_first_name`, `user_last_name`

Tradeoff:

- ✅Pro: Historical accuracy preserved forever
- ✅Pro: Faster queries (no JOINs needed)
- ✅Pro: Data integrity even if user/address deleted
- ❌Con: Data duplication
- ❌Con: More columns in orders/returns tables

---

4. Payment Processing - No Payment Table

Decision: NO dedicated `payments` table, use third-party processors (Stripe/PayPal)

Reasoning:

- NEVER store credit card numbers or CVV codes (PCI compliance nightmare)
- Payment processors handle all sensitive data
- We only need to track payment status and transaction reference

Implementation: Store in `orders` table:

- `payment_hash` - External transaction ID from Stripe/PayPal
- `payment_method` - Which service used ('stripe', 'paypal', 'apple_pay')
- `payment_status` - Current status ('pending', 'completed', 'failed', 'refunded')

Tradeoff:

- ✅Pro: PCI compliant by default
- ✅Pro: No security liability
- ✅Pro: Simpler architecture
- ❌Con: Dependent on third-party for payment details

---

5. Shipment Table - Kept Separate

Decision: Keep `shipments` as separate table instead of adding fields to `orders`

Reasoning:

- Supports multiple shipments per order (split fulfillment)
- Keeps orders table cleaner
- Allows complex shipping workflows if needed later

Tradeoff:

- ✅Pro: Flexible for future requirements
- ✅Pro: Cleaner separation of concerns
- ❌Con: Extra JOIN to get shipment status
- ❌Con: Might be overkill for simple store

Alternative Considered: Add `tracking_number`, `carrier`, `shipped_at`, `delivered_at` directly to orders

- Could work for simpler use cases with always 1 shipment per order

---

6. Calculated Fields - What We Store

Decision: Store calculated totals in `orders` table

Fields Stored:

- `total_discount` - Total discount applied
- `total_tax` - Tax charged
- `shipping_cost` - Shipping cost
- `total_amount` - Final amount paid

Reasoning:

- Orders are immutable financial/legal records
- Prices/taxes may change over time
- Refund calculations need original amounts
- Performance (no recalculation needed)

Tradeoff:

- ✅Pro: Historical accuracy for refunds/taxes
- ✅Pro: Faster reporting queries
- ✅Pro: Legal compliance (exact records)
- ❌Con: Data redundancy (could be calculated)

---

7. Guest Checkout Support

Decision: Support guest checkout with nullable `user_id` and `session_id`

Implementation:

- `carts.user_id` - Nullable, NULL for guests
- `carts.session_id` - Track anonymous users
- `orders.user_id` - Nullable for guest orders

Tradeoff:

- ✅Pro: Lower friction for first-time buyers
- ✅Pro: Better conversion rates
- ❌Con: More complex user tracking
- ❌Con: Need session management

Important Constraints

1. One Active Cart Per User

Need constraint: Only ONE active cart per logged-in user at a time

```sql
CREATE UNIQUE INDEX ON carts(user_id) WHERE status = 'active' AND user_id IS NOT NULL;
```

2. Password NOT Unique

Password should never have unique constraint - multiple users can have same password (with different salts)

3. Phone Type

Store phone as VARCHAR(20), not INT

- Handles country codes (+1)
- Handles formatting (555) 123-4567
- Preserves leading zeros

Optional Enhancements to Discuss

1. SKU Field in product_variants

- Needed for: Inventory systems, warehouse fulfillment, integrations
- Add: `sku VARCHAR(50) UNIQUE`

2. Product Variant Status

- Needed for: Hide variants without deleting them
- Add: `status VARCHAR(20) CHECK (status IN ('active', 'inactive', 'discontinued'))`

3. Restocking Tracking

- Needed for: Track if returned items added back to inventory
- Add to return_items: `is_restocked BOOLEAN DEFAULT false`

4. State/Province in Address

- Needed for: Shipping calculations, tax calculations (especially US)
- Add: `state_province VARCHAR(50)`

5. Promo Codes / Discounts

- Current: Only stores `total_discount` amount
- Missing: What promo code was used? When does it expire?
- Consider: Separate `promotions` table with codes, validity dates, usage limits

---

Cart Lifecycle Flow

Active Shopping:

```
1. User has cart (status='active')
2. Items in cart_items table
3. User modifies cart freely (add/remove/update)
```

Checkout:

```
1. Create order record (with user/address snapshot)
2. Copy cart_items → order_items (with product snapshot)
3. Process payment (Stripe/PayPal)
4. Update order payment_status = 'completed'
5. Mark cart status = 'order' (or delete)
6. Create new active cart for user
```

Cleanup:

```
1. Delete old abandoned carts (status='abandoned', >30 days old)
2. cart_items CASCADE deleted with cart
3. order_items remain intact (no CASCADE)
```

Questions for Team Discussion

1. **Do we need the shipments table** or can we simplify by adding tracking fields directly to orders?
2. **Guest checkout** - Do we want to support it from day 1 or add later?
3. **Promo codes** - Do we need them in v1? If yes, need to design promotions table.
4. **Product images** - Current: one image_url per variant. Do we need multiple images per product?
5. **Inventory reservation** - When do we decrement stock?
   - At checkout only (current approach)
   - When added to cart (soft reservation)
   - Something else?
6. **Cart cleanup** - Should we auto-delete abandoned carts? After how many days?

Database Tables Summary

Core Tables:

- `users` - Customer accounts
- `addresses` - Shipping/billing addresses
- `products` - Base product info
- `product_variants` - Size/color combinations with inventory
- `carts` - Shopping carts
- `cart_items` - Items in active carts
- `orders` - Completed purchases (with snapshots)
- `order_items` - Items in orders (with product snapshots)
- `shipments` - Tracking info
- `returns` - Return requests
- `return_items` - Specific items being returned

Key Relationships:

```
user → carts → cart_items → product_variant
user → orders → order_items (snapshot)
order → shipments
order → returns → return_items → order_items
product → product_variants
```

Next Steps

1. Review this document as a team
2. Discuss any tradeoffs or alternatives
3. Decide on optional enhancements (SKU, promo codes, etc.)
4. Finalize schema
5. Create database migration scripts
6. Begin implementation

Document Version: 1.0
Last Updated: 2025-11-02