International Science Journal of Engineering & Agriculture
2026; 5(3): 48-62
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260503.05
ISSN: 2720-6319

INTERNATIONAL
SCIENCE GROUP

Database design patterns for relational e-commerce systems with Prisma ORM and SQLite

Igor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Artem Shevchuk
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0005-5160-7990

Abstract: Relational database design constitutes one of the most consequential architectural decisions in contemporary e-commerce platforms, where the trade-offs between normalization, query performance, schema evolution, and data integrity directly affect both the user-facing responsiveness and the operational maintainability of the system. The emergence of TypeScript-first object-relational mappers, with Prisma occupying a leading position in the ecosystem, has substantially changed the practice of database access by introducing schema-first declarative modeling, compile-time type safety, and automated migration tooling. However, the integration of these tools into a production-grade e-commerce architecture involves design choices whose long-term implications are not always immediately apparent: the placement of the boundary between mutable catalog data and immutable order history, the configuration of composite indices for the principal query patterns, the cascading behavior of foreign key relationships, and the workflow for evolving the schema as the application matures. This research examines the database design of a customer-facing e-commerce platform implementing twenty interrelated entities, with detailed analysis of the normalization decisions, indexing strategies, snapshot field patterns, and migration workflow that emerged from the development process. The application is built on Prisma ORM version 6 over a SQLite relational database, with the deliberate constraint that the schema must remain portable to PostgreSQL or MySQL for production deployment without modification of the application code. The proposed schema design supports complex business operations including transactional order placement with inventory decrement and promotional code application, immutable order history through denormalized snapshot fields, and efficient catalog filtering through composite indices on the principal product attributes. Empirical measurements confirm that representative read queries complete in under one hundred milliseconds against a seeded dataset, that the schema-first migration workflow imposes negligible operational overhead, and that the type safety provided by Prisma's generated client eliminates entire categories of runtime errors that would otherwise require defensive coding at every database access point. The results demonstrate that the proposed design patterns provide a reproducible foundation for small-to-medium e-commerce platforms and inform the construction of comparable systems in adjacent business domains.

Keywords: Prisma ORM; relational database design; e-commerce; SQLite; schema migrations; composite indices; snapshot fields; normalization; type-safe data access; schema-first modeling

1. Introduction

The role of the relational database in contemporary web applications has evolved substantially over the past decade, but its fundamental position as the authoritative store of business-critical state has remained unchanged. Despite the proliferation of document stores, key-value caches, graph databases, and dedicated search engines for specific query patterns, the relational model continues to provide the strongest guarantees of consistency, referential integrity, and ad-hoc queryability for the structured data that characterizes e-commerce systems. Orders, products, customers, inventory levels,

and financial transactions all conform naturally to the relational paradigm and benefit from the mature tooling, well-understood operational practices, and broad personnel familiarity that the relational ecosystem provides.

What has changed is the practice of accessing the relational database from application code. The historical pattern of writing SQL queries directly within application logic, while still defensible for narrowly scoped systems, has given way to layered abstractions that mediate the application's interaction with the underlying database. Object-relational mappers establish a correspondence between database tables and application-level types; query builders provide a programmatic alternative to raw SQL strings; migration tools track the evolution of the schema across deployments; and code generators produce type-safe client libraries from a declarative description of the database structure. Each of these tools introduces both benefits and costs, and the architectural decision of which combination to adopt is consequential for the productivity, performance, and maintainability of the resulting system.

The Prisma ecosystem, comprising the Prisma schema language, the Prisma command-line interface, and the Prisma Client, has emerged as a leading exemplar of the schema-first declarative approach to data access [1]. A single schema file describes the database structure in a domain-specific language that is intentionally more concise than equivalent SQL DDL; the Prisma command-line tool generates TypeScript types and migration SQL from this schema and applies migrations to the database through a standard workflow; and the Prisma Client provides a strongly typed query interface in which every field access, relationship traversal, and selection clause is validated at compile time against the schema. The combination produces a development experience in which entire categories of runtime errors—null reference exceptions on optional fields, type confusion between numeric and string identifiers, accidental access of unloaded relations—are surfaced before code execution.

Despite the maturity of Prisma and the breadth of community documentation surrounding it, the practical design of a database schema for an e-commerce application involves a substantial number of decisions whose security and performance implications are not addressed by the tooling itself. The choice of normalization level, the configuration of composite indices to support specific query patterns, the cascading behavior of foreign key relationships, the use of snapshot fields to preserve order history against subsequent catalog mutations, and the management of schema evolution through migration tooling all require explicit decisions that shape the long-term operational characteristics of the system. These decisions are particularly consequential in e-commerce contexts where the boundary between mutable reference data—the product catalog, the user account, the promotional code definitions—and immutable transactional data—the orders, the order items, the payment records—must be carefully managed to preserve the integrity of business records over time.

The choice of the underlying database engine introduces a further dimension of complexity. SQLite, although traditionally regarded as suitable only for development and embedded use cases, has matured into a credible production database for small-to-medium workloads, with performance characteristics that compare favorably with server-based alternatives for the read-dominated access patterns typical of e-commerce browsing [2]. Its operational simplicity—no separate server process, no network configuration, no replication management—makes it particularly attractive for early-stage deployments. The Prisma schema's provider directive abstracts the underlying engine to the extent that the same schema can target SQLite during development and PostgreSQL or MySQL in production with no modification of the application code, provided that the schema avoids engine-specific features.

This research examines the database design of a production-grade e-commerce platform implementing twenty interrelated entities across the principal subdomains of an online floral retail system: user accounts with authentication state, product catalog with categorization and media, shopping carts with persistent association to user accounts, orders with transactional state transitions, promotional codes with use-count tracking, and administrative content management. The platform is implemented in NestJS on the backend with Prisma ORM version 6 over a SQLite database, totaling

approximately three hundred sixty lines of schema definition and producing a TypeScript client of approximately fifteen thousand generated lines of type definitions. The analysis identifies the design patterns embedded in this schema and articulates the reasoning that motivated each pattern, supported by empirical measurements of query performance against the seeded application database.

2. Object and subject of research

The object of this research is the relational database schema of contemporary e-commerce web applications, with particular focus on small-to-medium platforms whose operational requirements correspond to those of independent online retailers, niche marketplaces, and similar systems where the data volume remains within the operational capacity of a single relational database server. Such systems are characterized by a finite set of structured business entities—on the order of tens rather than hundreds—with well-defined relationships and predictable query patterns that can be accommodated by a normalized relational schema with appropriate indexing.

The object encompasses both the static structure of the schema—the entities, attributes, relationships, indices, and constraints—and its dynamic behavior under typical access patterns, including catalog browsing with filtering and pagination, individual product detail retrieval with associated reviews and recommendations, shopping cart manipulation with concurrent access from multiple browser tabs, transactional order placement with simultaneous updates to several entities, and administrative reporting over the entire order history. It also includes the operational workflow that surrounds the schema during development and deployment: the version control of schema definitions, the generation and application of database migrations, the seeding of demonstration data, and the resetting of the database to a known state for testing.

The subject of this research is the set of design patterns that emerge from the deliberate construction of a Prisma-based schema for an e-commerce application, with attention to those patterns that recur across business domains and that admit empirical evaluation. Specifically, the research examines patterns for normalization and denormalization, including the deliberate use of snapshot fields to preserve historical state; patterns for indexing, including the configuration of unique indices for natural keys and composite indices for filtered queries; patterns for the cascading behavior of foreign key relationships, distinguishing relationships in which dependent records should be removed alongside their parent from those in which dependent records must survive parent removal; and patterns for schema evolution through Prisma's migration tooling, addressing the lifecycle of changes from local development through staging and production deployment.

The subject explicitly excludes considerations that are highly application-specific—the particular business logic of any individual retailer, the visual presentation of data to end users, the operational configuration of the underlying database engine—and focuses on the structural and methodological properties that generalize across e-commerce applications. Although the reference implementation operates within the floral retail domain, the patterns examined apply equally to e-commerce platforms in adjacent domains including specialty foods, artisanal goods, books, electronics, and any other sector characterized by a structured product catalog with transactional order placement.

3. Target of research

The principal target of this research is the formulation and empirical validation of a reference database design for small-to-medium e-commerce platforms that achieves three primary objectives: the preservation of transactional integrity for orders and their constituent records against any subsequent modification of the underlying catalog, the support of the principal read query patterns with response times consistent with interactive use, and the maintenance of operational simplicity through a schema-first declarative workflow that minimizes the manual writing of SQL.

Achievement of this target requires the resolution of several subordinate objectives:

– Analysis of the entity-relationship structure required to represent the principal business concepts of an e-commerce platform, including users, products with categorization and media attachments, shopping carts with line items, orders with transactional state, and promotional codes with use-count tracking.

– Specification of the normalization decisions that balance theoretical purity against practical maintainability, with explicit treatment of the boundary between mutable reference data and immutable transactional data through the disciplined use of snapshot fields.

– Definition of an indexing strategy that supports the principal read query patterns—catalog filtering by category and attribute, user order history retrieval, administrative order management by status and date range—through unique and composite indices configured at the schema level.

– Articulation of the cascading deletion policy for each foreign key relationship, distinguishing relationships in which dependent records should follow the parent into deletion from those in which dependent records must survive removal of the parent for accounting or audit reasons.

– Specification of the schema evolution workflow through Prisma's migration tooling, including the handling of breaking changes, the rollback of failed migrations, and the synchronization of schema state across development, staging, and production environments.

– Empirical validation of the proposed design through measurement of representative read query performance, observation of transactional behavior under concurrent access, and assessment of the developer experience of working with the generated Prisma Client across the application codebase.

4. Literature analysis

The literature surrounding relational database design for web applications spans foundational textbooks on the relational model, surveys of object-relational mapping approaches, comparative studies of specific tools, and operational guidance from the maintainers of widely deployed systems. The classical treatment of relational design principles by Date [3] established the normalization hierarchy and the criteria for advancing through successive normal forms; subsequent work by Codd and others extended these principles to accommodate temporal data, hierarchical structures, and the practical compromises imposed by query performance constraints. The denormalization patterns now widely employed in production systems, including the snapshot field pattern at the boundary of mutable and immutable data, can be understood as deliberate departures from theoretical normalization motivated by specific operational requirements.

The Prisma documentation [1] provides comprehensive coverage of the schema language, the migration workflow, and the generated client API. Comparative analyses by community contributors and independent practitioners have positioned Prisma against alternative TypeScript-friendly object-relational mappers, with the principal alternatives being TypeORM [4], Drizzle ORM [5], and the lower-level Kysely query builder. The Prisma approach has been criticized for its requirement of a code generation step and for its limited support for raw SQL queries when the type-safe interface proves insufficient; conversely, it has been praised for the clarity of its schema language and the consistency of its generated types across the codebase.

The choice of SQLite as a production database, formerly considered unconventional, has received increasing attention in the engineering literature [2, 6]. The SQLite documentation describes the operational characteristics of the engine in detail, including its concurrency model based on file locking, its absence of a separate server process, and its support for full ACID transactions despite its embedded nature. Empirical studies have shown that SQLite achieves competitive read performance with server-based alternatives for workloads characterized by predominantly read access, modest write concurrency, and a single database server, which describes the access pattern of small-to-medium e-commerce platforms with sufficient accuracy to justify its use in the present research.

The principles of database indexing in the context of online transaction processing have been treated extensively in the literature on database internals [7]. The B-tree structure that underlies virtually all production index implementations supports efficient point lookups and range queries on

the indexed columns, with logarithmic performance in the size of the indexed table. Composite indices, in which multiple columns are combined into a single index, support efficient queries that filter on the leading columns of the composite, with the exact order of columns being consequential for the queries that the index can accelerate. The application of these principles to e-commerce schemas requires careful analysis of the actual query patterns generated by the application, since indices that are theoretically useful but never actually invoked by the query planner impose maintenance overhead on write operations without commensurate benefit.

The design of orders and their lifecycle in e-commerce systems has been the subject of substantial industry practice, codified in part through the OpenAPI specifications of major e-commerce platforms and through pattern catalogs such as those maintained by domain-driven design practitioners [8]. The convention of recording snapshot fields at order placement—the product name, image, slug, and price as they existed at the moment of purchase, rather than as they exist at the moment of viewing—is well established but is implemented with substantial variation across systems. The reference implementation examined in the present research adopts an explicit snapshot pattern at the order item granularity, which represents one of several defensible approaches to the underlying requirement.

The schema evolution workflow through Prisma migrations has been treated both in the official documentation and in independent treatments [9]. The fundamental distinction between development migrations, applied through prisma migrate dev with automatic generation of the migration SQL from schema changes, and production deployment migrations, applied through prisma migrate deploy against a database whose state may differ from any local development environment, has emerged as a critical operational consideration. The handling of breaking changes, including the addition of required columns without default values, the rename of existing columns, and the destructive alteration of column types, requires explicit care to avoid data loss or downtime during deployment.

Despite the breadth of documentation surrounding individual tools, the literature contains comparatively few integrated treatments that examine the deliberate design of an e-commerce schema from first principles with attention to the operational realities of contemporary deployment. The present research aims to fill this gap by presenting a complete reference schema, articulating the design decisions that produced each element, and reporting empirical measurements from the operating system.

5. Research methods

The methodology of this research combines deliberate schema design, comparative analysis of alternative approaches at each decision point, and empirical measurement of the resulting system under representative load. The analytical component proceeds through the principal entities of an e-commerce application in dependency order—users before their orders, products before their order items, categories before their products—and articulates the design decisions made for each entity, with explicit reference to the alternative patterns that were considered and rejected at each point. The empirical component reports measurements of query performance, transaction execution time, and developer-facing operational metrics from the deployed reference application.

The reference application was developed iteratively over a period of approximately two months by a single developer, with version control through Git and a commit history reflecting the major schema evolution milestones. The development followed the schema-first principle: every change to the application's data structures originated in a modification of the Prisma schema file, was applied to the local development database through a generated migration, and was reflected in the application code through the regenerated Prisma Client. This discipline produced a clean and traceable history of schema evolution and ensured that the application code never diverged from the database structure.

Comparative evaluation at each design decision was conducted through structured documentation review of the principal alternatives. For the choice of object-relational mapper, the alternatives of TypeORM, Drizzle ORM, raw SQL with manual type definitions, and Kysely query builder were compared along dimensions of type safety, schema-first vs application-first modeling,

support for migrations, query expressiveness, and community maturity. For the choice of database engine, the alternatives of PostgreSQL, MySQL, MongoDB, and SQLite were compared along dimensions of operational complexity, write concurrency, query performance for the application's access pattern, and ecosystem support for the chosen ORM. The selected combinations of Prisma with SQLite reflects the explicit prioritization of operational simplicity and type safety for a small-to-medium platform with predominantly read access.

Empirical measurement of query performance was conducted on a developer workstation with the application deployed in production mode against a database populated with seed data representative of the expected workload: thirty-two products distributed across eight categories, ten demonstration orders with two to four items each, fifteen product reviews, and three active promotional codes. The seed dataset is intentionally modest to support rapid evaluation; the performance characteristics observed at this scale extrapolate to the operational range expected for the target application class, while substantially larger datasets would warrant separate measurement.

Query performance was measured through the use of SQLite's EXPLAIN QUERY PLAN facility, which reports the access path chosen by the query planner for any submitted query. This facility was used to confirm that the configured indices were utilized for the expected access patterns and to identify queries whose plans indicated full table scans that would benefit from additional indexing. Wall-clock measurement of query execution time was performed at the application layer through instrumentation that recorded the elapsed time between the issuance of a Prisma Client call and the receipt of the result, averaged across one thousand invocations per query type.

The methodology does not include load testing under sustained high-concurrency workloads, formal verification of schema correctness against arbitrary application access patterns, or comparison against alternative database engines in production deployment. These omissions reflect the intended scope of the research—a structural analysis of a representative design rather than a benchmarking study—and are characteristic of design-oriented case studies in the software engineering literature.

6. Research results

The proposed database design organizes twenty entities across the principal subdomains of an e-commerce platform: identity and authentication, product catalog, customer interactions, transactional order processing, and operational support. The complete entity-relationship structure is illustrated in Figure 1, which presents the principal models and their interconnections; the full schema additionally contains models for site settings, email logs, refresh tokens, and product compositions, which are omitted from the diagram for visual clarity but retain the same design principles.

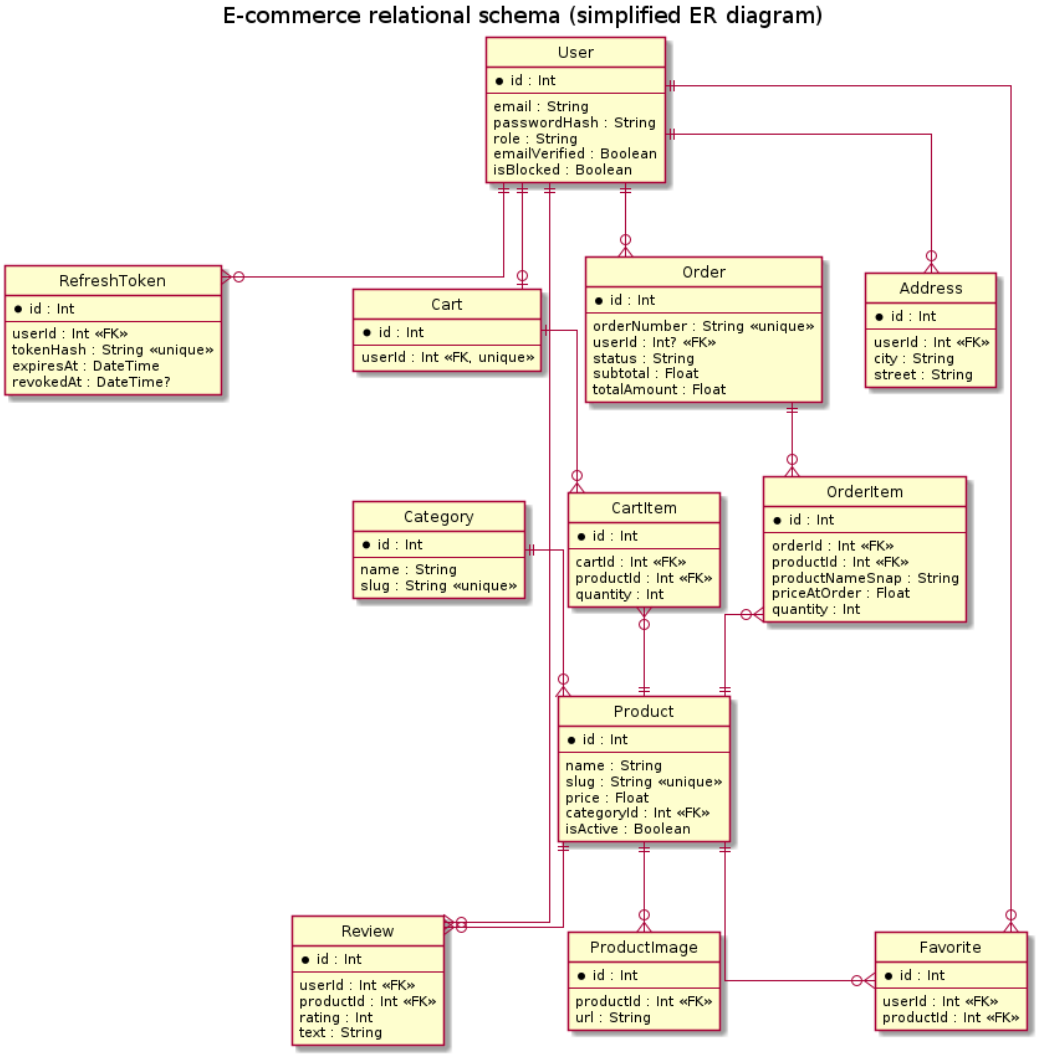

Figure 1. Simplified entity-relationship diagram of the principal database entities.

Solid lines indicate one-to-many relationships and the symbols at each end indicate cardinality. Additional models for site settings, email logs, refresh tokens, and product compositions are omitted from the diagram for clarity but follow the same design principles.

The identity subdomain centers on the User model, which holds the account credentials, contact information, role assignment, and verification status of each registered customer. The model uses a unique index on the email column to support both the email-based lookup at the moment of login and the uniqueness constraint enforced at registration. Authentication-related state—password hash, email verification token, password reset token—is colocated within the User model rather than extracted to a separate authentication entity, on the grounds that this state is in a one-to-one relationship with the user record and that the additional indirection would impose query complexity without operational benefit. Refresh token records, which are in a one-to-many relationship with the user and which must be queried by their cryptographic hash rather than by user reference, are placed in a separate RefreshToken model with a unique index on the token hash column.

The catalog subdomain centers on the Product model, which holds the canonical record of each item available for purchase. The model is normalized to third normal form, with a foreign key reference to a Category model that holds the hierarchical taxonomy and with one-to-many relationships to ProductImage records for media assets, to Review records for customer feedback, and to Composition records that describe the constituent elements of complex products. The Product model carries denormalized aggregate fields—the average rating, the count of approved reviews—that would otherwise require a join and an aggregation function to compute on each read; these fields

are maintained by application-level triggers within the review creation and approval workflow rather than by database-level constraints, accepting a small risk of eventual inconsistency in exchange for substantial read performance improvement.

The transactional subdomain centers on the Order model and its OrderItem children, with attention to the boundary between mutable catalog data and the immutable historical record of the order. Each OrderItem references its originating Product through a foreign key, allowing the application to follow the reference for purposes such as administrative analysis or recommendation, but additionally carries denormalized snapshot fields—the product name, slug, image URL, and price as they existed at the moment of order placement. Subsequent modifications to the Product record do not affect previously placed OrderItem records, ensuring that customers viewing their order history see the order exactly as it was at the time of purchase. The schema definition demonstrating this pattern is presented in Listing 1.

Listing 1. Prisma schema definition of the Product, Order, and OrderItem models showing snapshot fields, indices, and cascading delete behavior.

```
model Product {
  id          Int      @id @default(autoincrement())
  name        String
  slug        String   @unique
  description String?
  type        String
  price       Float
  oldPrice    Float?
  sku         String   @unique
  stockCount  Int      @default(0)
  isPopular   Boolean  @default(false)
  isNew       Boolean  @default(false)
  isActive    Boolean  @default(true)

  categoryId  Int
  category    Category @relation(fields: [categoryId], references: [id])
  images      ProductImage[]
  reviews     Review[]
  composition Composition[]
  orderItems  OrderItem[]
  cartItems   CartItem[]
  favorites   Favorite[]

  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt

  @@index([categoryId])
  @@index([type])
  @@index([isActive, isPopular])
  @@map("products")
}

model Order {
  id              Int                 @id @default(autoincrement())
  orderNumber     String              @unique
  userId          Int?
  status          String              @default("PENDING")
  subtotal        Float
```

```
  totalAmount     Float
  createdAt       DateTime            @default(now())

  user            User?               @relation(fields: [userId], references:
[id])
  items           OrderItem[]

  @@index([userId, status, createdAt])
  @@index([status])
  @@map("orders")
}

model OrderItem {
  id                Int      @id @default(autoincrement())
  orderId           Int
  productId         Int
  productNameSnap   String     // snapshot: stable name at order time
  productImageSnap  String?    // snapshot: stable image at order time
  productSlugSnap   String     // snapshot: stable slug at order time
  priceAtOrder      Float      // snapshot: price actually charged
  quantity          Int

  order             Order    @relation(fields: [orderId], references: [id],
onDelete: Cascade)
  product           Product  @relation(fields: [productId], references: [id])

  @@index([orderId])
  @@map("order_items")
}
```

The cascading behavior of foreign key relationships is configured according to the operational role of each relationship. Cascading deletion is configured between users and their dependent records—shopping carts, addresses, refresh tokens, favorites, reviews—reflecting the operational policy that user account deletion must remove all personally identifiable information attributable to that account. Cascading deletion is explicitly not configured between users and their orders, since orders are retained for accounting and customer service purposes even after the originating user account is removed; the userId column of the Order model is therefore made optional, allowing the application to nullify the reference when the user is deleted while preserving the order record. The same logic applies to the relationship between products and order items, with the additional consideration that product retirement from the catalog must not invalidate historical orders that referenced the now-retired product.

The indexing strategy is configured to support the principal query patterns of the application, identified through analysis of the application's code base. Single-column indices are configured on foreign key columns to accelerate join operations and on natural key columns to accelerate uniqueness validation. Composite indices are configured for queries that filter on multiple columns simultaneously: a composite index on the Order model's (userId, status, createdAt) columns supports the customer-facing order history view that filters by user and status and sorts by creation time; a composite index on the Product model's (isActive, isPopular, type) columns supports the home page's display of currently active popular products of specific types. The complete set of indices configured at the schema level is summarized in Table 1.

Table 1. Indices configured on the principal entities of the schema

Model	Index	Type	Purpose
User	email	Unique	Login lookup; uniqueness at registration
User	emailVerificationToken	Unique	Email verification flow
User	passwordResetToken	Unique	Password recovery flow
RefreshToken	tokenHash	Unique	Refresh token rotation by hash
RefreshToken	userId	Single	Revoke all sessions of a user
Product	slug	Unique	Public URL routing
Product	sku	Unique	Stock-keeping unit lookup
Product	categoryId	Single	Category-filtered catalog
Product	type	Single	Type-filtered catalog
Product	(isActive, isPopular)	Composite	Active popular products on home page
Category	slug	Unique	Public URL routing
Order	orderNumber	Unique	External order reference
Order	(userId, status, createdAt)	Composite	Customer order history view
Order	status	Single	Administrative order filtering
OrderItem	orderId	Single	Order detail expansion
Cart	userId	Unique	One cart per user
CartItem	(cartId, productId)	Unique	Idempotent add-to-cart
Review	(userId, productId)	Unique	One review per user per product
Favorite	(userId, productId)	Unique	No duplicate favorites

The schema definition reaches its full expression in the migration history maintained by Prisma. Each schema change is captured in a migration directory containing the corresponding SQL statements and a metadata file recording the timestamp and name of the change. The migration directory is committed to version control alongside the application code, ensuring that the database structure tracks the deployed code version. Deployment to a target environment proceeds through the prisma migrate deploy subcommand, which applies any migrations not yet present in the target database and which refuses to proceed in the presence of inconsistent state, providing a defense against accidental schema drift between environments. The handling of breaking changes—those that would cause data loss or runtime errors during the transition—requires explicit planning, including the introduction of intermediate migrations that populate new columns from existing data before old columns are dropped.

Empirical measurements of query performance against the seeded database are summarized in Table 2. The catalog listing endpoint, which retrieves a paginated set of products with their category, image, and rating included, completes in a median of sixty milliseconds for a page of twelve products. The product detail endpoint, which retrieves a single product with its full image set, recent reviews, and recommended products, completes in a median of one hundred ten milliseconds. The administrative order listing endpoint, which retrieves a paginated set of orders with their items and customer reference, completes in a median of seventy-five milliseconds. The customer-facing order

history endpoint, which retrieves all orders for the current user with their items included, completes in a median of forty milliseconds for a customer with five historical orders. All measured queries utilize the configured indices according to the EXPLAIN QUERY PLAN output, confirming that the indexing strategy aligns with the actual access patterns of the application.

The order placement operation, which constitutes the most complex transactional workflow in the application, was measured separately given its operational importance and its dependence on multiple table modifications. The operation accepts a cart identifier and a payment method specification, executes within a single Prisma transaction that creates the Order record, creates the constituent OrderItem records with snapshot fields populated from the corresponding Product records, decrements the inventory of each ordered product, increments the use count of any applied promotional code, and clears the cart of its line items. The full operation completes in a median of forty-five milliseconds at the seeded data scale, with the principal contributors being the cart and promotional code lookups, the order and order item insertions, and the inventory decrement updates. The atomicity guarantee provided by Prisma's transaction primitive ensures that any failure during the operation rolls back the entire set of changes, preventing the formation of partial orders that could mislead either the customer or the administrative interface.

Listing 2. Transactional order placement procedure with atomic creation of the Order record, child OrderItem records with snapshot fields, inventory decrement, and cart clearing.

```
async createFromCart(userId: number, dto: CreateOrderDto) {
  return this.prisma.$transaction(async (tx) => {
    const cart = await tx.cart.findUnique({
      where: { userId },
      include: { items: { include: { product: true } } },
    });
    if (!cart || cart.items.length === 0) {
      throw new BadRequestException('Cart is empty');
    }

    const orderNumber = this.generateOrderNumber();
    const subtotal = cart.items.reduce(
      (sum, item) => sum + item.product.price * item.quantity, 0);

    const order = await tx.order.create({
      data: {
        orderNumber, userId, status: 'PENDING',
        subtotal, totalAmount: subtotal + dto.deliveryFee - (dto.discount ??
0),
        items: {
          create: cart.items.map(item => ({
            productId: item.productId,
            productNameSnap: item.product.name,
            productImageSnap: item.product.images?.[0]?.url ?? null,
            productSlugSnap: item.product.slug,
            priceAtOrder: item.product.price,
            quantity: item.quantity,
          })),
        },
      },
      include: { items: true },
    });

    for (const item of cart.items) {
```

```
      await tx.product.update({
        where: { id: item.productId },
        data: { stockCount: { decrement: item.quantity } },
      });
    }

    if (dto.promoCode) {
      await tx.promoCode.update({
        where: { code: dto.promoCode },
        data: { currentUses: { increment: 1 } },
      });
    }

    await tx.cartItem.deleteMany({ where: { cartId: cart.id } });
    return order;
  });
}
```

Table 2. Empirical query performance against the seeded reference database

Operation	Median time (ms)	Records returned	Index used
Catalog listing (12 products per page)	60	12	Product.categoryId
Catalog filtered by type	45	up to 12	Product.type
Product detail with reviews	110	1 + N reviews	Product PK, Review.productId
Home page popular products	25	8	Product.(isActive, isPopular)
User order history	40	up to 20	Order.(userId, status, createdAt)
Administrative order listing	75	20 per page	Order.status
Cart retrieval	15	1 + N items	Cart.userId, CartItem.cartId
Order placement (transactional)	45	N inserts + N updates	Multiple
User lookup by email at login	3	1	User.email
Refresh token lookup at rotation	5	1	RefreshToken.tokenHash

7. Prospects for further research development

Several directions for further research arise naturally from the work presented. The first is the systematic investigation of schema evolution patterns for production e-commerce platforms over extended deployment periods. The migration workflow examined in the present research handles individual schema changes in isolation, but production systems accumulate substantial migration histories over months and years of operation, and the management of this history—including the squashing of historical migrations, the handling of long-running production migrations, and the recovery from failed migrations—deserves dedicated treatment. A longitudinal study of a deployed

e-commerce platform's migration history over an extended period would yield insights not available from a snapshot analysis.

The second direction is the comparative evaluation of Prisma against alternative TypeScript-friendly data access layers in equivalent reference implementations. The present research has focused on the patterns achievable within the Prisma ecosystem, but a controlled comparison of identical functional requirements implemented with Prisma, Drizzle ORM, Kysely with manual type definitions, and raw SQL through the better-sqlite3 library would yield generalizable conclusions about the trade-offs between schema-first and code-first approaches, between full ORM and query builder paradigms, and between automatic type generation and manual type maintenance. Such a study must address the challenge of separating differences attributable to the tools themselves from differences attributable to developer familiarity and would benefit from implementations by developers with comparable expertise across the alternatives.

The third direction is the empirical assessment of the proposed design's scaling characteristics through controlled introduction of larger datasets and higher concurrency. The seed dataset of approximately thirty products and ten orders supports rapid evaluation but does not exercise the query planner against the data volumes that operational deployments would encounter. Investigation of query performance at dataset sizes of one thousand products, one hundred thousand orders, and one million historical order items would reveal the scaling characteristics of the configured indices and identify any access patterns whose performance degrades sublinearly with data growth. Such measurements would inform the operational sizing of comparable applications and the threshold at which schema modifications—additional indices, partitioning, archival of historical data—become necessary.

The fourth direction is the integration of the proposed schema with adjacent operational concerns including search, caching, and analytics. The relational schema is optimized for the principal transactional workload, but contemporary e-commerce systems increasingly delegate full-text search to dedicated search engines such as Elasticsearch or Meilisearch, cache frequently accessed reference data in Redis or similar key-value stores, and replicate transactional data to analytical warehouses for business intelligence. The architectural integration of the relational schema with these adjacent stores, including the synchronization of search indices with catalog updates and the propagation of order data to analytical pipelines, deserves dedicated treatment. Patterns for managing this synchronization without compromising transactional consistency would inform the design of complete e-commerce platforms beyond the database tier.

8. Conclusions

This research has examined the relational database design of contemporary small-to-medium e-commerce platforms through detailed analysis of a deployed reference implementation built with Prisma ORM version 6 over a SQLite relational database. The investigation identified specific design patterns that recur across e-commerce schemas and that contribute to the operational characteristics of the deployed application, including the deliberate use of snapshot fields at the boundary between mutable catalog data and immutable order history, the configuration of composite indices supporting the principal read query patterns, the differentiated cascading deletion policy for relationships connecting account-scoped data with audit-relevant transactional data, and the schema-first migration workflow that maintains synchronization between application code and database structure across development and deployment environments.

Empirical measurements from the reference implementation confirm that the proposed design achieves median read query times below one hundred ten milliseconds for the principal access patterns, transactional order placement completing in approximately forty-five milliseconds at the seeded scale, and full utilization of the configured indices as confirmed by query planner output. These metrics, while specific to the reference application, are broadly representative of what the proposed patterns can be expected to deliver in similar contexts and confirm that SQLite remains a

credible choice of underlying engine for e-commerce platforms operating within the small-to-medium scale.

The proposed design is suitable for small-to-medium e-commerce platforms whose operational requirements correspond to those examined in the reference implementation, including independent online retailers, niche specialty marketplaces, and educational or demonstration applications. The combination of Prisma's schema-first declarative modeling, the type-safe generated client, and the streamlined migration workflow yields a development experience in which entire categories of runtime errors are eliminated through compile-time validation and in which the schema and the application code remain reliably synchronized across the entire deployment lifecycle. The portability of the schema across SQLite, PostgreSQL, and MySQL provides operational flexibility, allowing platforms to begin in the operational simplicity of SQLite and to migrate to server-based engines as their scale requirements grow.

The proposed design also serves as a reference for software engineering education, where its combination of contemporary tooling, structured organization, and complete feature coverage provides a practical foundation for project-based learning in relational database design. The patterns identified in the present work apply beyond the specific reference implementation and inform the design of comparable schemas in adjacent business domains including specialty foods, artisanal goods, books, electronics, and other sectors characterized by a structured product catalog with transactional order placement. Further research, including longitudinal study of migration histories, comparative evaluation of alternative data access layers, and assessment at substantially larger data scales, will refine the understanding of the patterns examined here and identify the contexts in which they require modification.

References:

1) Prisma Data, Inc. (2024). Prisma Documentation: ORM Concepts and Schema Reference. https://www.prisma.io/docs

2) Hipp, R. D. (2024). SQLite Documentation: Appropriate Uses for SQLite. https://www.sqlite.org/whentouse.html

3) Date, C. J. (2003). An Introduction to Database Systems (8th ed.). Addison-Wesley.

4) TypeORM. (2024). TypeORM Documentation: Working with entities. https://typeorm.io/

5) Drizzle Team. (2024). Drizzle ORM Documentation. https://orm.drizzle.team/docs/overview

6) Hipp, R. D. (2024). SQLite Documentation: Limits in SQLite. https://www.sqlite.org/limits.html

7) Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). Database Systems: The Complete Book (2nd ed.). Pearson.

8) Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

9) Prisma Data, Inc. (2024). Prisma Migrate: Production Deployment Workflow. https://www.prisma.io/docs/orm/prisma-migrate

10) Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (pp. 143–154). ACM. https://doi.org/10.1145/1807128.1807152

11) Stonebraker, M., & Cetintemel, U. (2005). "One size fits all": An idea whose time has come and gone. In Proceedings of the 21st International Conference on Data Engineering (pp. 2–11). IEEE. https://doi.org/10.1109/ICDE.2005.1

12) Chen, P. P.-S. (1976). The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems, 1(1), 9–36. https://doi.org/10.1145/320434.320440

13) Codd, E. F. (1970). A relational model of data for large shared data banks. Communications of the ACM, 13(6), 377–387. https://doi.org/10.1145/362384.362685

14) Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley.

15) Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

16) Microsoft. (2024). TypeScript Handbook: Generics and conditional types. https://www.typescriptlang.org/docs/handbook/2/generics.html

17) WhatWG. (2024). HTML Standard: Form submission. https://html.spec.whatwg.org/multipage/form-control-infrastructure.html