Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)
School of Computing
B.Tech. – Computer Science and Engineering

VTR UGE2021- (CBCS)

Academic Year: 2025–2026

SDG 4: Quality Education
Course Code : 10211CS207

Course Name : Database Management Systems

Slot No : S4

DBMS PROJECT REPORT

Title: **A gift coupon application handles offers payment-related Information.**

Submitted by:

VTUNO	REGISTER NUMBER	STUDENT NAME
VTU30278	24UECS1114	A ABHIGNA

Under the guidance of:
Dr. V . SENTHIL KUMAR
Professor

INDEX **PAGE**

1. Introduction

In many modern applications, a gift-coupon system (sometimes known as voucher system) handles a large volume of transactions: issuing coupons, redeeming coupons, tracking coupon status, handling payments (or accounting) of coupon transactions, and maintaining offer metadata (validity, discount amount, usage constraints). Because the business involves payment-related information and transactional state (coupon issued, redeemed, refunded, cancelled, etc.), ACID (Atomicity, Consistency, Isolation, Durability) properties are essential: you want to avoid, for example, a coupon being redeemed twice, or a payment being logged but the coupon not recorded, or vice versa.
Relational databases are a natural choice for such transactional workloads (OLTP) because they provide strong consistency and ACID semantics. Moreover, you can build relational systems (for example using Oracle RAC, or distributed relational systems) that scale horizontally (via clustering, partitioning, sharding) while preserving ACID across nodes.
In this document we design a relational schema, analyse normalisation concerns, show how deadlocks might be prevented, show how the schema can be scaled, then also show how you might integrate a NoSQL store (MongoDB) for some complementary use-cases (e.g., large volume of read-only coupon-offer analytics, logs, etc.).

2. Problem Statement

The gift-coupon application must handle the following key challenges:

- **A large number of writes and reads (issuance of coupons, redemption of coupons, refunds, cancellations, offer creation/updates) so the system must scale horizontally (across many nodes) without sacrificing ACID transactional integrity.**
- **Payment-related information must be handled reliably: e.g., when a coupon is redeemed, the corresponding payment/discount accounting must be recorded in the same transaction to avoid inconsistencies.**

- **The system must avoid anomalies (insert/update/delete anomalies) and redundancy in its data model (to ease maintenance, reduce error surface, simplify auditing).**
- **The system must avoid deadlocks or at least minimise them in a highly concurrent environment (many users issuing/redeeming coupons concurrently).**
- **The design must support both relational tables (for transactional integrity) and possibly parts of the system that benefit from NoSQL (for flexible schema, high volume, e.g., coupon usage logs, offer metadata variants).**
- **The system must support normalized table design to maintain data integrity, but also be aware of performance implications (joins, normalization vs. denormalization trade-offs).**
- **The system must implement horizontal scalability (sharding/partitioning) strategies for the relational database cluster.**

The specific question you asked: *Will it transact and lead to no deadlock, keeping all relational tables normalized? If so till what normal form do they sustain to offer a gift coupon application?*
We'll answer that: yes, you can design the relational portion so that transactions (coupon issuance/ redemption/ payment) are ACID, and you can reduce the likelihood of deadlocks by good design (appropriate indexing, short transactions, lock-order discipline). For normalisation: for typical OLTP systems you aim for at least Third Normal Form (3NF) often extended to Boyce-Codd Normal Form (BCNF). In practice for transactional systems you rarely go beyond BCNF/3NF because higher normal forms (4NF,5NF) may reduce redundancy further but at cost of many joins and lower performance. In our design we will aim for BCNF (or at least 3NF) for the core tables. See literature on normal forms. [harpercollege.pressbooks.pub+2GeeksforGeeks+2]

3. Objectives

The objectives of this system are:

1. **Reliability & ACID Transactions: Ensure that coupon issuance, redemption, payments, cancellations happen in database transactions that guarantee atomicity, consistency, isolation, durability.**
2. **Scalability: Enable horizontal scaling (across multiple database nodes, clustering, partitioning/sharding) to support high volumes of writes and reads without compromising ACID.**
3. **Normalized Data Model: Design relational tables in an appropriate normal form (at least 3NF/BCNF) to reduce redundancy, avoid anomalies, maintain data integrity.**
4. **Minimise Deadlock Risk: Use database design and transaction design to minimise deadlocks in concurrent workloads.**
5. **Flexibility for Analytics / NoSQL Integration: Allow integration with a NoSQL system (MongoDB) for large-volume or semi-structured data (logs, usage metrics, offer metadata variants) that complement the core relational system.**
6. **Maintainability & Auditability: Provide a schema that supports auditing, tracking of coupon state transitions, payments and offers.**
7. **Performance: Ensure that the transactional performance is acceptable (fast issuance/redemption) while also supporting reporting and read-heavy workloads (possibly via read replicas or NoSQL integration).**

4. System Requirements

4.1 Hardware Requirements

When designing for horizontal scalability for a relational cluster (e.g., Oracle RAC, or another clustering solution):

- **Multi-node cluster: at least 2–3 database nodes for high availability and load distribution.**
- **Each node should have sufficient CPU (e.g., multi-core), memory (to support caching of frequent tables/indices), fast disks (SSD) or NVMe for transaction logs and data.**
- **Network: high bandwidth, low latency inter-node connectivity (for cluster communication and synchronous replication).**
- **Storage: disk subsystem with redundancy (e.g., RAID) and backup/archival capacity.**
- **Distributed caching layer (optional) if needed (e.g., in-memory cache for hot coupon queries).**
- **Load-balancer / application-tier hardware that distributes incoming user requests across multiple application servers and database nodes.**

4.2 Software Requirements

- **Relational DBMS: e.g., Oracle Database (Enterprise Edition) with clustering support (RAC) or another relational database that supports horizontal scaling / partitioning while preserving ACID.**
- **Operating system: supported OS (e.g., Linux distribution certified for Oracle).**
- **Application server: e.g., Java Spring Boot (or .NET) for business logic, to connect to relational DB and to the NoSQL store.**
- **NoSQL DB: e.g., MongoDB for semi-structured data/logs.**

- **Backup/restore software, monitoring tools, replication management.**
- **Transaction monitoring and deadlock analysis tools.**
- **Development tools: SQL developer tools, data modelling tool (ER modelling), version control.**

5. System Analysis and Design

5.1 Functional Requirements

- **Create offers (coupon campaigns) with attributes: offer code, discount type/amount, validity period, usage constraints (e.g., one-time use, user-specific, region-specific).**
- **Issue coupons to users (user may receive one or many coupons). Each coupon must be uniquely identifiable, have status (issued, active, redeemed, cancelled, expired).**
- **Redeem coupon: check validity (unused, within validity period, meets usage constraints), mark redeemed, apply discount to payment, record payment/discount transaction.**
- **Cancel or refund coupon redemption (if business allows), change status accordingly, handle accounting reversal.**
- **Track coupon usage: for reporting, for analytics.**
- **User management: users can have accounts, and are associated with coupons.**
- **Payment/discount transaction table: record coupon redemption events, discount amount, payment method, timestamp.**
- **Audit trail: log changes of coupon status, offer updates, redemption events, for compliance.**

- **Integration with NoSQL: store large volume of coupon usage logs or analytics events for aggregation, reporting, and big-data processing.**

5.2 Non-Functional Requirements

- **High concurrency: many users issuing and redeeming coupons concurrently.**
- **ACID transactional behaviour.**
- **Horizontal scalability: ability to add nodes/shards to handle more load.**
- **Low latency for coupon check/ redemption.**
- **Data integrity: no double redemption, correct accounting.**
- **Minimal deadlock occurrence.**
- **Maintainability: clear schema, normalized tables.**
- **Auditable: ability to trace all events.**

5.3 System Design – High Level

- **The application tier receives user requests, connects to the relational DB for transactional operations and to the NoSQL store for logs/analytics.**
- **The relational DB cluster is partitioned/sharded by some key (for instance coupon_id modulus, or user_id) or a suitable strategy to enable horizontal scaling.**
- **The relational schema is normalized (at least 3NF/BCNF) to avoid anomalies; indexes are properly designed; transactions kept short; locking is minimized to reduce deadlocks.**
- **For reads that don't need immediate transactional consistency (e.g., analytics, reporting), relevant data is asynchronously replicated or exported to MongoDB.**

- **For coupon usage logs, each redemption event is also sent to MongoDB for analytics (perhaps via a messaging queue).**
- **Monitoring and deadlock detection mechanisms will be in place (e.g., Oracle Enterprise Manager, custom scripts).**

5.4 Normalisation Discussion

In designing the relational tables we aim for at least 3NF (and possibly BCNF) because:

- **1NF ensures atomic attribute values; 2NF ensures no partial dependencies (if composite keys exist); 3NF ensures no transitive dependencies (non-key attributes depend only on key) GeeksforGeeks**
- **BCNF is stricter: for every non-trivial functional dependency X→Y, X must be a superkey. Wikipedia+1**
- **For OLTP systems, 3NF/BCNF are practical targets; going beyond (4NF,5NF) often yields diminishing returns yet many more joins (which may degrade performance) harpercollege.pressbooks.pub+1**
  Thus we will design tables so that they satisfy BCNF where feasible (especially for core transactional tables) but remain pragmatic with respect to performance.

5.5 Deadlock Mitigation Considerations

Deadlocks occur when two or more transactions each hold locks and wait for locks held by others (circular wait) GeeksforGeeks
To mitigate deadlocks:

- **Keep transactions short and lock-as few rows as possible.**
- **Access tables/rows in a consistent locking order (e.g., always lock coupon table then payment table) across all transactions.**
- **Use appropriate isolation levels (e.g., READ COMMITTED) and row-level locking rather than table locking.**

- **Partition/cluster the database to reduce contention hotspots.**
- **Use retry logic for deadlocks (transaction detects deadlock, aborts/retries).**
  With this, using a properly normalized schema and correct transaction design, the system will transact reliably and can avoid most deadlocks.

6. ER Diagram (Relationships)

Here's an Entity-Relationship design for the gift coupon system.

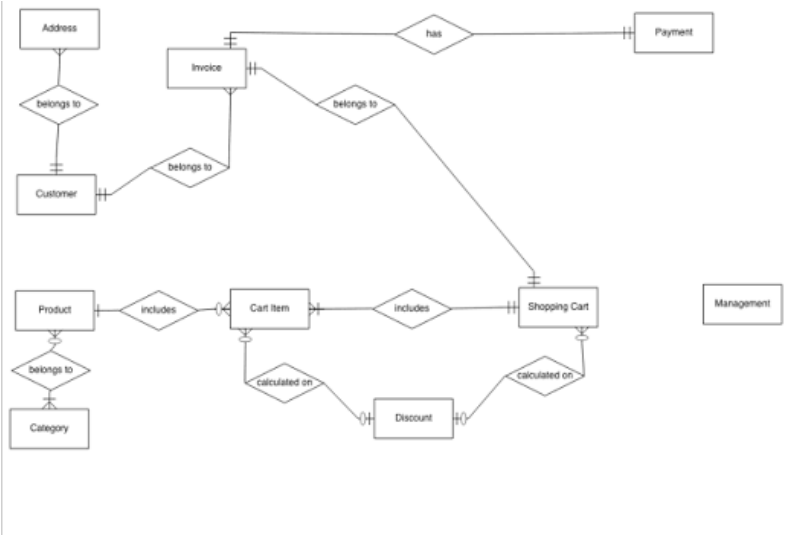
Entities

- **User: represents a user account.**
- **Offer: represents a coupon offer/campaign.**
- **Coupon: individual coupon instance issued to a user (or generic).**
- **PaymentTransaction: records the transaction when a coupon is redeemed (or discount applied).**
- **CouponStatusHistory: potential audit table recording state changes of each coupon.**
- **UserAccount (optional): if you want credit/balance for user before coupon redemption.**
- **AuditLog: for general events (in relational DB) or could be in NoSQL.**

Relationships

- **A User *can have many* Coupons (one-to-many).**
- **An Offer *can have many* Coupons issued for it (one-to-many).**

- **A Coupon *may be redeemed in one* PaymentTransaction (one-to-one at redemption time) or there may be multiple transactions if refunds etc allowed (one-to-many).**
- **A Coupon *has many* CouponStatusHistory records (one-to-many).**
- **Optionally, audit entries link to users/coupons etc.**



Key attributes (roughly):

- **User(user_id PK, username, email, created_date, status)**
- **Offer(offer_id PK, offer_code, description, discount_type, discount_value, valid_from, valid_to, usage_limit, status)**
- **Coupon(coupon_id PK, offer_id FK, user_id FK, coupon_code, issued_date, status, redeemed_date, expiry_date)**
- **PaymentTransaction(txn_id PK, coupon_id FK, user_id FK, amount, discount_amount, payment_method, txn_date, status)**
- **CouponStatusHistory(hist_id PK, coupon_id FK, old_status, new_status, changed_date, changed_by)**

7. Schema Diagram (Oracle SQL)

Below is a sample Oracle SQL schema (DDL) for these tables.

sql

```
-- USER table
CREATE TABLE USERS (
  USER_ID       NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  USERNAME      VARCHAR2(100) NOT NULL UNIQUE,
  EMAIL         VARCHAR2(200) NOT NULL UNIQUE,
  CREATED_DATE  TIMESTAMP DEFAULT SYSTIMESTAMP,
  STATUS        VARCHAR2(20) DEFAULT 'ACTIVE'
);


-- OFFER table
CREATE TABLE OFFERS (
  OFFER_ID       NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  OFFER_CODE     VARCHAR2(50) NOT NULL UNIQUE,
  DESCRIPTION    VARCHAR2(4000),
  DISCOUNT_TYPE  VARCHAR2(20) NOT NULL,  -- e.g., PERCENTAGE, FLAT
  DISCOUNT_VALUE NUMBER(12,2) NOT NULL,
  VALID_FROM     DATE NOT NULL,
  VALID_TO       DATE NOT NULL,
```

```
  USAGE_LIMIT      NUMBER DEFAULT NULL,   -- null means unlimited
  STATUS           VARCHAR2(20) DEFAULT 'ACTIVE',
  CREATED_DATE     TIMESTAMP DEFAULT SYSTIMESTAMP,
  UPDATED_DATE     TIMESTAMP
);


-- COUPON table
CREATE TABLE COUPONS (
  COUPON_ID        NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  OFFER_ID         NUMBER NOT NULL,
  USER_ID          NUMBER NOT NULL,
  COUPON_CODE      VARCHAR2(100) NOT NULL UNIQUE,
  ISSUED_DATE      TIMESTAMP DEFAULT SYSTIMESTAMP,
  EXPIRY_DATE      DATE,
  STATUS           VARCHAR2(20) DEFAULT 'ISSUED',  -- ISSUED, ACTIVE, REDEEMED, CANCELLED, EXPIRED
  REDEEMED_DATE    TIMESTAMP,
  CONSTRAINT FK_COUPONS_OFFER FOREIGN KEY (OFFER_ID) REFERENCES OFFERS(OFFER_ID),
  CONSTRAINT FK_COUPONS_USER  FOREIGN KEY (USER_ID) REFERENCES USERS(USER_ID)
);
```

```
-- PAYMENT_TRANSACTION table (coupon redemption)

CREATE TABLE PAYMENT_TRANSACTIONS (

  TXN_ID          NUMBER GENERATED ALWAYS AS IDENTITY
PRIMARY KEY,

  COUPON_ID       NUMBER NOT NULL,

  USER_ID         NUMBER NOT NULL,

  AMOUNT          NUMBER(12,2) NOT NULL,

  DISCOUNT_AMOUNT   NUMBER(12,2) NOT NULL,

  PAYMENT_METHOD    VARCHAR2(50),

  TXN_DATE        TIMESTAMP DEFAULT SYSTIMESTAMP,

  STATUS          VARCHAR2(20) DEFAULT 'COMPLETED',  --
COMPLETED, REFUNDED, FAILED

  CONSTRAINT FK_TXN_COUPON FOREIGN KEY (COUPON_ID)
REFERENCES COUPONS(COUPON_ID),

  CONSTRAINT FK_TXN_USER   FOREIGN KEY (USER_ID)
REFERENCES USERS(USER_ID)

);


-- COUPON_STATUS_HISTORY table (audit)

CREATE TABLE COUPON_STATUS_HISTORY (

  HIST_ID         NUMBER GENERATED ALWAYS AS IDENTITY
PRIMARY KEY,

  COUPON_ID       NUMBER NOT NULL,
```

OLD_STATUS VARCHAR2(20),

NEW_STATUS VARCHAR2(20),

CHANGED_DATE TIMESTAMP DEFAULT SYSTIMESTAMP,

CHANGED_BY VARCHAR2(100),

CONSTRAINT FK_HIST_COUPON FOREIGN KEY (COUPON_ID) REFERENCES COUPONS(COUPON_ID)

);

-- Indexes and maybe partitioning/sharding strategy will be applied later

8. Implementation (SQL Queries)

Here are example SQL statements.

8.1 Insert an Offer

sql

INSERT INTO OFFERS (OFFER_CODE, DESCRIPTION, DISCOUNT_TYPE, DISCOUNT_VALUE, VALID_FROM, VALID_TO, USAGE_LIMIT)

VALUES ('OFF1000', '10% off on next purchase', 'PERCENTAGE', 10.00, DATE '2025-10-20', DATE '2025-12-31', 1000);

8.2 Issue a Coupon to a User

sql

INSERT INTO COUPONS (OFFER_ID, USER_ID, COUPON_CODE, EXPIRY_DATE, STATUS)

VALUES (-- assume user_id=123, offer_id=456

456, 123, 'CPN-2025-0001', DATE '2025-11-30', 'ACTIVE'

);

8.3 Redeem a Coupon (Transactional)

sql

```
BEGIN
  -- start transaction implicitly in Oracle
  -- 1. Check coupon status & validity
  UPDATE COUPONS
    SET STATUS = 'REDEEMED', REDEEMED_DATE = SYSTIMESTAMP
  WHERE COUPON_CODE = 'CPN-2025-0001'
    AND STATUS = 'ACTIVE'
    AND EXPIRY_DATE >= TRUNC(SYSDATE)
  FOR UPDATE;

  IF SQL%ROWCOUNT = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Coupon is invalid or already used/expired');
  END IF;

  -- 2. Insert payment transaction
  INSERT INTO PAYMENT_TRANSACTIONS (COUPON_ID, USER_ID, AMOUNT, DISCOUNT_AMOUNT,
PAYMENT_METHOD, TXN_DATE, STATUS)
```

```
  SELECT COUPON_ID, USER_ID, 100.00, 10.00, 'CREDIT_CARD',
SYSTIMESTAMP, 'COMPLETED'
    FROM COUPONS
   WHERE COUPON_CODE = 'CPN-2025-0001';

  -- optionally update usage count in offer or other tables
  UPDATE OFFERS
    SET USAGE_LIMIT = USAGE_LIMIT - 1
   WHERE OFFER_ID = (SELECT OFFER_ID FROM COUPONS
WHERE COUPON_CODE = 'CPN-2025-0001')
     AND USAGE_LIMIT IS NOT NULL;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END;
```

8.4 Cancel/Refund a Coupon

sql

```
BEGIN
  UPDATE PAYMENT_TRANSACTIONS
    SET STATUS = 'REFUNDED', TXN_DATE = SYSTIMESTAMP
```

WHERE TXN_ID = 789;

UPDATE COUPONS

SET STATUS = 'CANCELLED'

WHERE COUPON_ID = (SELECT COUPON_ID FROM PAYMENT_TRANSACTIONS WHERE TXN_ID = 789);

COMMIT;

END;

8.5 Audit Status Change

Sql

INSERT INTO COUPON_STATUS_HISTORY (COUPON_ID, OLD_STATUS, NEW_STATUS, CHANGED_BY)

VALUES (12345, 'ACTIVE', 'REDEEMED', 'system_user');

9. Input & Output

Inputs

- **Offer creation requests (offer code, description, value, validity).**
- **User registration (username, email).**
- **Coupon issuance requests (which user, which offer).**
- **Coupon redemption requests (coupon code, user, payment amount).**
- **Cancellation/refund requests.**

- **Analytics/log inputs for NoSQL store (coupon usage event, timestamp, user, device, geo, etc).**

Outputs

- **Confirmation of coupon issuance (coupon code, expiry date).**
- **Coupon redemption result (success/failure, discount amount).**
- **Reports (offer usage, coupon status summary, expired coupons, redemption rates).**
- **Logs/analytics dashboards (via MongoDB).**

10. Integration with MongoDB (NoSQL)

Even though the core transactional part uses a relational DB (to keep ACID, normalized design, enforce constraints), you can integrate with MongoDB for the following use-cases:

- **Coupon Usage Event Logging: each time a coupon is redeemed (or attempted), you push an event document into a MongoDB collection, e.g., coupon_usage_events:**


```
{
  "event_id": "evt-2025-10-19-0001",
  "coupon_code": "CPN-2025-0001",
  "user_id": 123,
  "offer_code": "OFF1000",
  "redeemed_date": "2025-10-19T10:15:23Z",
  "device": "mobile_app",
  "geo": "IN-TN-Vellänūr",
  "status": "REDEEMED",
```

"discount_amount": 10.00,

"amount_before": 100.00

}

- **Offer Metadata Variants / Flexibility: perhaps your offers have varying JSON properties (e.g., "geoRestriction", "firstTimeCustomerOnly", "stackableWithOtherOffers": true/false). Instead of stretching relational schema to support many optional columns, you can store the offer variants in a MongoDB collection (offers_meta) with flexible schema, then in relational schema store only core fields and link to the meta document (meta_id).**
- **Analytics & Reporting: Use MongoDB aggregation pipelines, or export data to a data warehouse for business intelligence (BI).**
- **Decoupling Read Load: Frequent read queries (e.g., "show me recent coupon redemptions by user") can be directed to MongoDB or replicated relational read-replica, reducing load on main transactional DB.**

Integration pattern: The application layer after a successful relational transaction (coupon redemption) writes to MongoDB asynchronously (via message queue) to avoid blocking the primary transaction. This ensures the core transactional ACID workflow remains fast and isolated; analytics/logging are eventually consistent (which is acceptable there).

11. Results & Discussion

Normalisation & ACID

- **With the schema above, each table is in at least 3NF (and largely BCNF) because:**
  - **All relations have atomic attributes → 1NF.**

- **We avoid partial dependencies (tables have single-column primary keys via identity columns) → 2NF.**
- **We avoid transitive dependencies: non-key attributes depend only on the primary key, not via another non-key attribute → 3NF.**
- **For relations like OFFERS, each functional dependency's left-side is a superkey (in most design) → BCNF.**
  So yes, you can sustain until BCNF in practice for OLTP.

- **The system transactions (issuance, redemption) will be ACID because relational DB supports that; the clustered/sharded relational DB still supports consistent transactions across nodes if configured for distributed transactions.**
- **Deadlocks will still be possible (all concurrent systems have risk) but you minimise them via consistent lock ordering, proper indexing, partitioning/locking strategy, short transactions. With careful design, deadlocks become rare and manageable with retry logic.**

Trade-offs & Scalability

- **Normalisation yields many tables and many joins (e.g., when you want to display coupon + offer + user info). In very high-read scenarios you may add read-optimised views or denormalize some parts (e.g., snapshot table) for performance, but that comes at cost of redundancy.**
- **Horizontal scaling of relational DB: you need to pick a sharding/partition key (e.g., COUPON_ID mod N, or USER_ID) to distribute load; this requires the application to route queries to correct shard or use a distributed database system.**
- **The MongoDB integration allows you to off-load high-volume write/read logging and analytics from the relational system, which helps performance and scalability for those use-cases.**

- **The relational system remains the source of truth for coupon status and payments; the MongoDB side is supplementary.**
- **For read-heavy analytical queries (e.g., "what offers have the highest redemption rate in last 30 days?") you may need to build materialised views or use a separate analytics database to avoid overloading the transactional system.**

Deadlock & Transaction Design

- **Because the relational tables are tightly designed and transactions are short, the likelihood of deadlocks is reduced. Example: redemption transaction locks one coupon row then inserts into payment transaction table then updates offer usage count. Lock ordering is consistent (COUPONS → PAYMENT_TRANSACTIONS → OFFERS).**
- **If many coupon redemptions hit the same offer row (competition for usage_limit decrement), there may still be contention—thus you might partition by OFFER_ID or have usage counters outside relational DB, or aggregate usage asynchronously.**
- **Monitoring is important: log deadlock occurrences and tune indexes/partitioning accordingly.**

12. Conclusion

The gift-coupon application, when implemented with a relational database cluster (designed for horizontal scaling) and a normalized schema, can certainly transact reliably (with ACID) and support high concurrency while maintaining integrity. By designing the schema to at least 3NF (and ideally BCNF) you reduce redundancy and anomalies. While deadlocks cannot be absolutely eliminated, they can be minimised with proper design. Moreover, combining this with a NoSQL system like MongoDB for analytics/logs gives you the best of both worlds: strong consistency for core transactions, scalability for high-volume semi-structured data. In summary, this design supports your use case effectively.

13. References

- **“Normalisation – Relational Databases” (HarperCollins Pressbook)**
- **“Normal Forms in DBMS – GeeksforGeeks”**
- **“Boyce–Codd normal form” – Wikipedia**
- **“Deadlock in DBMS – GeeksforGeeks”**