HOMEWORK ASSIGNMENT 11 – CHAPTER 23

1. **Problem 23.27:**

Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 23.3 and the fragments at site 1 are as shown in Figure 3.6. For each of the following queries, show at least two strategies of decomposing and executing the query. Under what conditions would each of your strategies work well?

a) For each employee in department 5, retrieve the employee name and the names of the employee's dependents.
b) Print the names of all employees who work in department 5 but who work on some project not controlled by department 5.

Figure 23.3:

Figure 23.3
Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 1)]).

(a) Employees in Department 5

G1

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G2

Essn	Pno	Hours
333445555	10	10.0

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

Essn	Pno	Hours
333445555	20	10.0

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(b) Employees in Department 4

G4

Essn	Pno	Hours

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

Essn	Pno	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

Essn	Pno	Hours
987654321	20	15.0

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(c) Employees in Department 1

G7

Essn	Pno	Hours

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

Essn	Pno	Hours

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

Essn	Pno	Hours
888665555	20	Null

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

Figure 3.6:

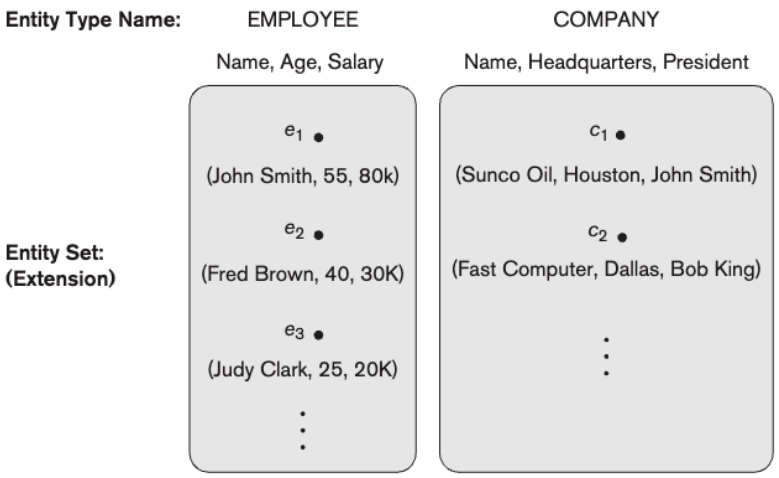

Figure 3.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

➔ Solution:

I. Retrieve the employee's name and names of the employee's dependents for each employee in department 5.

Given Information:

- Employee and Department relationships are depicted in the fragments at sites 1, 2, and 3.

a) Query Broadcast –

Decomposition:

- Retrieve all employees from department 5 from the Employee table.
- Join this with the Dependent table based on the Employee ID.

Execution:

- Broadcast the filter condition Department = 5 to all sites hosting fragments of Employee.
- Collect the results from these sites and execute a join with Dependent at a central site.

Conditions for Effectiveness:

- Works well if the number of employees in department 5 is relatively small, reducing the amount of data transfer.

b) Fragment Query and Local Join –

Decomposition:

- Locate and query only the fragment of Employee containing department 5 at the site where it's stored.
- Execute the join locally with Dependent at that site.

Execution:

- Transfer only the results to the requesting user.

Conditions for Effectiveness:

- Effective if fragments are vertically partitioned by department, and local processing reduces inter-site communication.

II. Retrieve names of all employees in department 5 who work on projects not controlled by department 5.

a. Distributed Semi-Join –

Decomposition:

- Select employees from department 5.
- Join with the Projects table to find projects they work on.
- Filter out projects controlled by department 5.

Execution:

- Semi-join to determine project IDs controlled by department 5.
- Distribute project control data to sites and compute locally.
- Final filtering and merging at a central site.

Conditions for Effectiveness:

- Reduces the size of intermediate results transferred across the network.

b. Centralized Query on Project-Control Relationship –

Decomposition:

- Collect department control data centrally.
- Execute joins centrally to match employees, projects, and controlling departments.

Execution:

- Transfer all relevant fragments to a central site and run the query there.

Conditions for Effectiveness:

- Suitable when network transfer costs are low or if project and department data are small.

2. Problem 23.28:

Consider the following relations:

BOOKS(Book#, Primary_author, Topic, Total_stock, $price)
BOOKSTORE(Store#, City, State, Zip, Inventory_value)
STOCK(Store#, Book#, Qty)

Total stock is the total number of books in stock, and Inventory value is the total inventory value for the store in dollars.

a. Give an example of two simple predicates that would be meaningful for the BOOKSTORE relation for horizontal partitioning.
b. How would a derived horizontal partitioning of STOCK be defined based on the partitioning of BOOKSTORE?
c. Show predicates by which BOOKS may be horizontally partitioned by topic.
d. Show how the STOCK may be further partitioned from the partitions in (b) by adding the predicates in (c)

➔ Solution:

a) Example of Two Simple Predicates for Horizontal Partitioning of BOOKSTORE:

Horizontal partitioning divides the BOOKSTORE table into subsets based on meaningful predicates. Here are two example predicates:

1. State = 'CA' (all bookstores located in California)
2. City = 'New York' (all bookstores located in New York City)

b) Derived Horizontal Partitioning of STOCK Based on BOOKSTORE:

The STOCK table has a Store# column that serves as a foreign key to BOOKSTORE(Store#). The derived horizontal partitioning of STOCK follows the partitioning of BOOKSTORE. For example:

i) From BOOKSTORE partition where State = 'CA':
SELECT * FROM STOCK WHERE Store# IN (SELECT Store# FROM BOOKSTORE WHERE State = 'CA');

ii) From BOOKSTORE partition where City = 'New York':
SELECT * FROM STOCK WHERE Store# IN (SELECT Store# FROM BOOKSTORE WHERE City = 'New York');

c) Horizontal Partitioning of BOOKS by Topic:

To partition the BOOKS table by Topic, meaningful predicates could include:

i) Topic = 'Fiction' (books with a fiction topic)
ii) Topic = 'Science' (books with a science topic)
iii) Topic = 'History' (books with a history topic)

d) Further Partitioning STOCK Based on BOOKSTORE and BOOKS:

To further partition the STOCK table, combine the predicates from (b) and (c). This creates partitions by store location and book topic. For example:

i) STOCK partition for stores in California and fiction books:

```
SELECT * FROM STOCK
WHERE Store# IN (SELECT Store# FROM BOOKSTORE WHERE State = 'CA')
  AND Book# IN (SELECT Book# FROM BOOKS WHERE Topic = 'Fiction');
```

ii) STOCK partition for stores in New York and science books:

```
SELECT * FROM STOCK
WHERE Store# IN (SELECT Store# FROM BOOKSTORE WHERE City = 'New York')
  AND Book# IN (SELECT Book# FROM BOOKS WHERE Topic = 'Science');
```

This results in a multi-level partitioning:

- Level 1: Geographical partitioning of STOCK based on BOOKSTORE.
- Level 2: Thematically partitioned STOCK data within each geographical partition based on BOOKS.

Review Questions:-

3. **Problem 23.17**

How is the decomposition of an update request different from the decomposition of a query? How are guard conditions and attribute lists of fragments used during the decomposition of an update request?

➔ Solution:

i) **Differences Between Decomposing an Update Request and a Query:**

 Purpose:

- Query Decomposition: Focuses on retrieving data. The goal is to efficiently access data from fragments and combine results to satisfy the query.
- Update Decomposition: Focuses on modifying data. The goal is to ensure changes are made correctly across all relevant fragments without violating constraints or consistency.

Fragment Selection:

- Query Decomposition: Only needs to access fragments that contain relevant data, optimizing the retrieval process.
- Update Decomposition: Must determine which fragments to update and ensure all affected fragments are identified, as missing an update may lead to inconsistency.

Data Propagation:

- Query Decomposition: Combines results from different fragments to produce a unified result.
- Update Decomposition: Ensures that updates propagate to all fragments where the data is stored or derived.

Constraints Handling:

- Query Decomposition: Constraints are typically checked during retrieval to ensure correct results.
- Update Decomposition: Constraints like primary keys, foreign keys, and other integrity rules must be enforced actively to avoid invalid data states.

Concurrency and Atomicity:

- Query Decomposition: May optimize for parallel reads across fragments.
- Update Decomposition: Needs to ensure atomicity and consistency of updates, often requiring locks or transactions.

ii) **Use of Guard Conditions and Attribute Lists in Update Decomposition:**

Guard Conditions:

- Definition: Conditions associated with fragments that specify which rows of a table are stored in the fragment (e.g., State = 'CA' for a fragment of the BOOKSTORE table).
- Role in Updates:
    - Guard conditions determine which fragments are relevant for a given update.
    - Example: For an update to State = 'CA', only fragments with the guard condition State = 'CA' are accessed and modified.

Attribute Lists:

- Definition: The set of attributes present in a fragment.
- Role in Updates:

- Attribute lists ensure updates modify only the relevant columns in the appropriate fragments.
- Example: If a fragment contains only City and State attributes of BOOKSTORE, an update to Inventory_value would not affect this fragment.

iii)Example: Update Decomposition with Guard Conditions and Attribute Lists:

```
UPDATE BOOKSTORE
SET Inventory_value = Inventory_value + 10000
WHERE State = 'CA';
```

Decomposition Process:

1. **Guard Conditions:**
   - Identify fragments where State = 'CA' (e.g., Fragment1: State = 'CA').
   - Only fragments matching this condition are selected for update.
2. **Attribute Lists:**
   - Check if the Inventory_value attribute is present in the fragments.
   - If the fragment does not store Inventory_value, the update skips it.
3. **Execution:**
   - Apply the update to the relevant fragments:

```
UPDATE Fragment1
SET Inventory_value = Inventory_value + 10000
WHERE State = 'CA';
```

4. **Propagation (if derived fragments exist):**
   - If other fragments derive their data from the updated fragment, ensure updates are propagated appropriately.

4. Problem 23.20

Discuss the two-phase commit protocol used for transaction management in a DDBMS. List its limitations and explain how they are overcome using the three-phase commit protocol.

➔ Solution:

i)Two-Phase Commit Protocol (2PC) in Distributed Database Management Systems (DDBMS)

The **Two-Phase Commit (2PC)** protocol is a consensus algorithm used in distributed systems to ensure all participating nodes (or sites) in a transaction either commit or abort the transaction together. This guarantees atomicity across a distributed database.

Phases in 2PC

1. **Phase 1: Prepare**
    - The **coordinator** sends a PREPARE request to all participating nodes.
    - Each participant performs local checks (e.g., constraints, locks, and resource availability) and votes:
        - If the participant can commit, it responds YES to the coordinator.
        - If it cannot, it responds NO and immediately aborts the transaction.
    - The coordinator waits for all responses.
2. **Phase 2: Commit or Abort**
    - If all participants vote YES:
        - The coordinator sends a COMMIT message to all participants.
        - Each participant performs the commit and acknowledges back to the coordinator.
    - If any participant votes NO:
        - The coordinator sends an ABORT message to all participants.
        - Each participant rolls back any changes and releases locks.

Limitations of 2PC

1. **Blocking Problem:**
    - If the coordinator crashes after sending the PREPARE messages but before sending the COMMIT or ABORT message, the participants remain in a blocked state, holding locks and waiting indefinitely.
2. **Single Point of Failure:**
    - The coordinator is a single point of failure. If it crashes, the entire protocol halts until it recovers.
3. **Message Overhead:**
    - Requires multiple rounds of communication, making it slow for large-scale systems.
4. **Limited Fault Tolerance:**
    - Cannot handle scenarios where participants fail during the PREPARE or COMMIT phases.

ii)Three-Phase Commit Protocol (3PC)

The **Three-Phase Commit (3PC)** protocol extends 2PC by introducing an additional phase to reduce the blocking problem and improve fault tolerance. It ensures no participant is left in an uncertain state even if a failure occurs.

Phases in 3PC

1. **Phase 1: Can Commit (Prepare Phase)**
    - The coordinator sends a CAN COMMIT message to all participants.
    - Participants check local readiness and reply YES or NO.

2. **Phase 2: Pre-Commit (Intermediate Phase)**
    - If all participants reply YES, the coordinator sends a PRE-COMMIT message to participants.
    - Participants perform all necessary operations (e.g., write logs, prepare data) but do not make the changes permanent.
    - Participants acknowledge receipt of the PRE-COMMIT.
    - If any participant replies NO in Phase 1, the coordinator sends an ABORT message, and all participants roll back.
3. **Phase 3: Commit**
    - If all participants acknowledge the PRE-COMMIT:
        - The coordinator sends a COMMIT message, and participants finalize the transaction.
    - If a participant fails to acknowledge the PRE-COMMIT, the coordinator sends an ABORT.

<u>Advantages of 3PC Over 2PC:</u>

1. **Non-Blocking:**
    - No participant remains in a blocked state waiting indefinitely because the intermediate phase allows participants to know the global state (commit or abort) without the coordinator.
2. **Improved Fault Tolerance:**
    - If the coordinator crashes, participants can still reach a consensus based on the intermediate state (e.g., PRE-COMMIT acknowledgment logs).
3. **Decoupling Commit Decision:**
    - The pre-commit phase ensures participants do not commit prematurely, avoiding partial commits.

Limitations of 3PC:

1. **Increased Message Overhead:**
    - The additional phase increases communication costs, making it slower for systems with low failure probability.
2. **Complexity:**
    - More complex to implement compared to 2PC, requiring careful handling of failure scenarios and timeouts.

5. Problem 23.24:

What are the main challenges facing a traditional DDBMS in the context of today's Internet applications? How does cloud computing attempt to address them?

➔ Solution:

Traditional DDBMS systems were designed for relatively stable and controlled environments. With the rise of Internet-scale applications, they face several challenges:

1. Scalability:

- **Challenge:** Traditional DDBMS systems struggle to scale horizontally to handle massive amounts of data and requests typical in modern applications like social media, e-commerce, and IoT.
- **Example:** A surge in user activity (e.g., during a flash sale or viral event) can overwhelm a traditional DDBMS.

2. High Availability and Fault Tolerance:

- **Challenge:** Ensuring availability and recovery from failures in a globally distributed system is difficult.
- **Example:** Internet applications require 24/7 uptime, but traditional DDBMSs often rely on a single master node or synchronous replication, which can lead to downtime during failures or network partitions.

3. Global Distribution:

- **Challenge:** Users of Internet applications are distributed globally, and traditional DDBMS systems are not optimized for low-latency access across multiple geographic regions.
- **Example:** A user in Asia accessing a database hosted in North America experiences high latency.

4. Heterogeneous Workloads:

- **Challenge:** Modern applications handle a mix of transaction processing, analytical queries, and unstructured data (e.g., images, logs).
- **Example:** Traditional DDBMSs are optimized for structured data but struggle to handle diverse workloads efficiently.

5. Dynamic Schema and Unstructured Data:

- **Challenge:** Internet applications frequently evolve, requiring schema changes or support for semi-structured and unstructured data, which traditional relational databases are not built to handle.
- **Example:** Adding new fields to a relational database can require downtime or costly migrations.

6. Cost Management:

- **Challenge:** Deploying and managing a distributed database across multiple locations is resource-intensive and expensive.
- **Example:** Maintaining hardware and ensuring redundancy is costly for organizations without dedicated infrastructure teams.

7. Performance Bottlenecks:

- **Challenge:** Traditional systems often face bottlenecks due to strict ACID compliance and centralized transaction coordination.
- **Example:** Global transactions across distributed nodes can lead to high latencies.

How Cloud Computing Addresses These Challenges

Cloud computing offers solutions by leveraging scalable, elastic, and globally distributed infrastructure:

1. Scalability:

- **Solution:** Cloud platforms provide horizontal scaling through database sharding and distributed storage solutions.
- **Examples:**
    - Amazon Aurora scales seamlessly across regions.
    - Google Spanner uses horizontal scaling to handle massive workloads.

2. High Availability and Fault Tolerance:

- **Solution:** Cloud providers offer built-in redundancy, multi-zone deployments, and automated failover.
- **Examples:**
    - Microsoft Azure's geo-replication ensures data availability even during regional outages.
    - Amazon RDS automatically replaces failed instances.

3. Global Distribution:

- **Solution:** Cloud databases like AWS DynamoDB and Google Spanner replicate data across multiple regions, optimizing for low-latency access near users.
- **Examples:**
    - DynamoDB’s global tables ensure sub-millisecond response times for users worldwide.
    - Cloudflare's edge network reduces latency for globally distributed users.

4. Support for Heterogeneous Workloads:

- **Solution:** Cloud databases offer specialized solutions for different workloads, such as transactional, analytical, and unstructured data processing.
- **Examples:**
    - AWS Redshift for analytics.
    - MongoDB Atlas for document-oriented storage.

5. Dynamic Schema and Support for Unstructured Data:

- **Solution:** NoSQL databases and schema-less designs are natively supported in cloud environments.
- **Examples:**
    - MongoDB and Firebase allow flexible data models.
    - Amazon S3 stores unstructured data like videos, logs, and backups.

6. Cost Management:

- **Solution:** Cloud services operate on a pay-as-you-go model, reducing upfront infrastructure costs.
- **Examples:**
    - Azure Cosmos DB charges based on data and throughput usage.
    - AWS RDS provides cost-efficient reserved instances.

7. Performance Optimization:

- **Solution:** Modern cloud databases balance consistency and availability using tunable consistency models and advanced caching.
- **Examples:**
    - Google Spanner offers a globally consistent database using TrueTime.
    - Amazon ElastiCache improves read performance with in-memory caching.

CHAPTER 24 REVIEW QUESTIONS

1. **Problem 24.5**

What is the CAP theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?

➔ Solution:

The **CAP Theorem**, proposed by Eric Brewer, states that in a distributed database system, it is impossible to simultaneously guarantee all three of the following properties:

1. **Consistency (C):**
    - Every read receives the most recent write or an error.
    - All replicas of the data reflect the same value at any given time.
2. **Availability (A):**
    - Every request receives a (non-error) response, regardless of the state of any individual node in the system.
    - The system remains operational even under node failures.
3. **Partition Tolerance (P):**
    - The system continues to function despite network partitions (communication breakdown between nodes in the system).
    - No amount of message loss or delay between nodes halts the system.

Trade-Off in CAP

- **Partition Tolerance** is essential in distributed systems because network failures are unavoidable.
- This forces a trade-off between **Consistency** and **Availability**:

- Systems choosing **Consistency** prioritize correctness but may sacrifice availability during partitions.
- Systems choosing **Availability** ensure responses to requests but may serve stale or inconsistent data.

Importance of Properties in NoSQL Systems

NoSQL databases are often used in scenarios where traditional relational databases struggle, such as handling high volumes of data, global distribution, and unstructured formats. The priorities in NoSQL systems vary depending on the specific use case.

1. Partition Tolerance (Most Important)

- Partition tolerance is **non-negotiable** in distributed systems, including NoSQL, due to the inevitability of network failures.
- Examples:
  - A NoSQL system like Cassandra or DynamoDB is designed to operate in geographically distributed environments, where partitions can occur.
  - Partition tolerance ensures these databases remain operational during network failures.

2. Availability

- NoSQL databases often prioritize **availability** over consistency.
- This is critical for systems requiring low latency and continuous operation, even at the cost of serving slightly stale data.
- Examples:
  - DynamoDB and Cassandra use eventual consistency, ensuring low-latency responses even during partitioned states.
  - Social media feeds, shopping cart systems, and IoT devices prioritize availability to maintain user experience.

3. Consistency

- While many NoSQL systems sacrifice strict consistency (strong consistency), some support tunable consistency levels:
  - **Eventual Consistency:** Guarantees that all replicas will eventually converge to the same state (e.g., DynamoDB, Cassandra).
  - **Strong Consistency:** Guarantees up-to-date data for every read (e.g., Google Spanner).
- Applications needing accurate data, like banking or inventory systems, may choose strong consistency with reduced availability.

NoSQL Systems and the CAP Theorem:-

NoSQL Database	**CAP Properties Prioritized**	**Examples**
Cassandra	Availability + Partition Tolerance	High availability for IoT, analytics
DynamoDB	Availability + Partition Tolerance	Eventual consistency for scalable applications
MongoDB	Availability + Partition Tolerance	Schema flexibility for web/mobile apps

NoSQL Database	CAP Properties Prioritized	Examples
Google Spanner	Consistency + Partition Tolerance	Strong consistency for financial applications
HBase	Consistency + Partition Tolerance	Hadoop ecosystem for batch processing

2. **Problem 24.6**

What are the similarities and differences between using consistency in CAP versus using consistency in ACID?

➔ Solution:

Definition of Consistency in CAP and ACID:-

1. **Consistency in CAP:**
   - Ensures that all nodes in a distributed system see the same data at the same time.
   - It guarantees that a read always reflects the latest write, or returns an error if the data cannot be guaranteed as consistent.
2. **Consistency in ACID:**
   - Ensures that a database transaction transforms the database from one valid state to another valid state while adhering to predefined rules, constraints, and integrity requirements.
   - It prevents violations of constraints (e.g., foreign keys, unique constraints).

Similarities:-

1. **Goal of Correctness:**
   - Both concepts focus on maintaining the correctness of the system's state.
   - **CAP Consistency** ensures uniformity across distributed replicas.
   - **ACID Consistency** ensures compliance with integrity constraints in a single database.
2. **Dependency on Other Properties:**
   - **CAP Consistency** often interacts with availability and partition tolerance in distributed systems, requiring trade-offs.
   - **ACID Consistency** depends on atomicity, isolation, and durability to ensure that transactions are either fully applied or not at all.
3. **Role in Distributed Systems:**
   - Both CAP and ACID consistency address challenges in distributed or transactional systems where failures and concurrent operations occur.

Differences:-

Aspect	Consistency in CAP	Consistency in ACID
Scope	Focused on distributed system replicas and data synchronization.	Concerned with a single database instance and transactional integrity.
When Applied	Applies globally across nodes in distributed systems.	Applies locally within a single database or transaction boundary.
Trade-Offs	In CAP, consistency is traded off with availability during network partitions.	In ACID, consistency is non-negotiable; it is strictly enforced in every transaction.
Definition	Ensures all nodes see the same data (latest write or error).	Ensures compliance with integrity constraints, e.g., foreign key and unique constraints.
Failure Handling	May temporarily allow stale or unavailable data to prioritize availability or partition tolerance.	Rolls back any transaction that would violate constraints to preserve a valid state.
Use Cases	Used in distributed systems like NoSQL databases.	Used in traditional relational databases and transactional systems.
Performance Impact	Consistency may result in latency due to synchronization between distributed nodes.	Enforcing consistency constraints can increase transaction execution time, especially under high contention.

Example Scenarios:-

CAP Consistency Example:

- In a distributed NoSQL database like Cassandra:
    - If a write operation is performed on one node, CAP consistency ensures that all other nodes reflect this change before responding to a read request.
    - Trade-off: The system may become unavailable during network partitions to preserve consistency.

ACID Consistency Example:

- In a relational database like MySQL:
    - A transaction to transfer money between accounts ensures that the total balance remains unchanged (constraint enforcement).
    - Trade-off: Transactions are rolled back if a constraint is violated, ensuring the database always remains valid.

3. Problem 24.7

What are the data modeling concepts used in MongoDB? What are the main CRUD operations of MongoDB?

➔ Solution:

MongoDB, as a **NoSQL database**, employs a schema-less, document-oriented model that differs from traditional relational databases. Its data modeling concepts revolve around flexibility, scalability, and performance.

Key Data Modeling Concepts

1. **Document-Oriented Model:**
   - Data is stored in BSON (Binary JSON) format as **documents**.
   - Each document is analogous to a row in a relational database but can have a flexible structure.
   - Example:

```
{
  "_id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "orders": [
    {"order_id": 101, "amount": 250},
    {"order_id": 102, "amount": 450}
  ]
}
```

2. **Collections:**
   - Documents are grouped into **collections**, similar to tables in relational databases.
   - Collections do not enforce a fixed schema, allowing documents with different structures to coexist.
3. **Embedded Data:**
   - MongoDB supports **embedded documents**, enabling hierarchical and nested structures.
   - Reduces the need for complex joins and improves query performance.
   - Example: An orders field in a user document containing an array of embedded order details.
4. **References:**
   - MongoDB allows **manual references** between documents to represent relationships (similar to foreign keys).
   - Example: A user_id field in an orders collection referencing the _id field in the users collection.
5. **Schema Flexibility:**
   - Unlike relational databases, MongoDB does not require a predefined schema. Fields can be added or omitted dynamically.
6. **Denormalization:**

    - MongoDB often encourages denormalization (storing related data together) to optimize read performance, as joins are not natively supported.
7. **Indexing:**
    - MongoDB supports indexes on fields within documents to speed up query performance.
8. **Sharding and Partitioning:**
    - MongoDB natively supports horizontal scaling using sharding, where data is partitioned across multiple servers based on a shard key.

Main CRUD Operations in MongoDB

CRUD stands for **Create, Read, Update, and Delete**, the primary operations performed on a database.

1. Create:

- Inserts a new document into a collection.
- **Command:**

```
db.collection.insertOne({
  name: "Alice",
  age: 30,
  email: "alice@example.com"
});
```

- **Example:** Insert multiple documents:

```
db.collection.insertMany([
  { name: "Bob", age: 25 },
  { name: "Charlie", age: 35 }
]);
```

2. Read:

- Queries documents from a collection.
- **Command**:
  db.collection.find({ age: { $gt: 25 } });
- **Example:** Find all documents in a collection:
  db.collection.find();
- Projection to limit fields:
  db.collection.find({ age: { $gt: 25 } }, { name: 1, _id: 0 });

3. Update:

- Modifies existing documents in a collection.
- **Command:**
    - Update one document:

```
db.collection.updateOne(
  { name: "Alice" }, // Filter
  { $set: { age: 31 } } // Update operation
);
```

- Update multiple documents:
  db.collection.updateMany(
    { age: { $lt: 30 } },
    { $set: { status: "young" } }
  );

4. **Delete:**

- Removes documents from a collection.
- **Command:**
  - Delete one document:
    db.collection.deleteOne({ name: "Alice" });
  - Delete multiple documents:
    db.collection.deleteMany({ age: { $lt: 30 } });