CBCS SCHEME

USN | | | | | | | | | |

21CS745

Seventh Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
NOSQL Database

Time: 3 hrs. Max. Marks: 100

Note: ***Answer any FIVE full questions, choosing ONE full question from each module.***

Module-1

1 a. Discuss the key difference between NOSQL and Relational databases. **(10 Marks)**

b. Provide strategies for optimizing data models to improve application performance. **(10 Marks)**

OR

2 a. Explain impendence mismatch problem in the context of data storage, and how does it affect application development. **(10 Marks)**

b. Describe the concept of schemaless databases and their benefits. **(10 Marks)**

Module-2

3 a. Explain the concepts of Master – Slave and Peer- to – peer replication in distributed databases. **(10 Marks)**

b. Discuss the importance of version stamp in esuring consistency across multiple nodes in a distributed database. **(10 Marks)**

OR

4 a. Discuss the challenges associated with achieving update consistency in a distributed database system. **(10 Marks)**

b. Discuss read consistency in distributed databases, considering factors such as staleness and isolation levels. **(10 Marks)**

Module-3

5 a. Provide an example of composing Map-Reduce calculations to process and Analyze data. **(10 Marks)**

b. Discuss the scalability characteristics of key-value databases. **(10 Marks)**

OR

6 a. Discuss key features of key value stores and their advantages. **(10 Marks)**

b. Describe the basic structure of data in a key value databases. **(10 Marks)**

Module-4

7 a. Explain fundamental principles of document database. **(10 Marks)**

b. Discuss the importance of SEO (Search Engine Optimization) in the context of blogging platforms. **(10 Marks)**

OR

8 a. Provide examples of common query operations performed on document databases and explain their significance. **(10 Marks)**

b. Explain the importance of event logging in web applications with examples. **(10 Marks)**

Module-5

9 a. Discuss the key features of graph databases that make them suitable for handling connected data. **(10 Marks)**

b. Identify scenarios where using a graph database may not be appropriate what are the limitations. **(10 Marks)**

OR

10 a. Describe how transactions are handled in graph database. What are the ACID properties are implemented. **(10 Marks)**

b. Provide examples of complex queries that can be efficiency executed in graph database. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

VTU EXAMINATION DEC-2024/JAN-2025
SCHEME OF EVALUATION

Sub:	**NOSQL DATABASE**					Sub Code:	**21CS745**	Branch:	**ISE**
Exam Date:	**20/01/2025**	Duration:	**3 Hrs**	Max Marks:	**100**	Sem/Sec:	**VII/ A, B & C**		**OBE**

Answer any **FIVE FULL** Questions choosing **ONE** full question from each **Module**

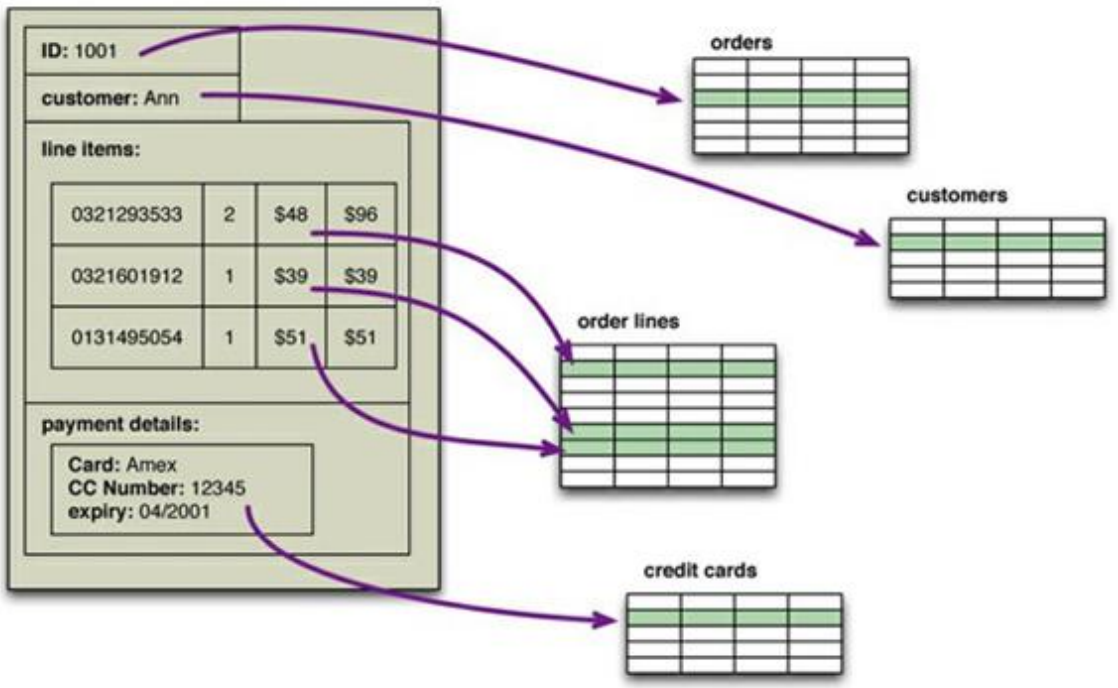
		MARKS	CO	RBT
	MODULE-1			
1.a	Discuss the key difference between NOSQL and Relational databases. **Scheme: Minimum 10 Aspects with differences for each carries 5+5 marks** **Solution:**	[10]	3	L2

Aspect	SQL (Relational)	NoSQL (Non-Relational)
Data Model	Structured, table-based with fixed schemas and relationships	Flexible data models (document, key-value, column-family, graph)
Schema	Rigid; must be predefined with strict structure	Schema-less or flexible; allows dynamic and unstructured data
Scalability	Vertical scaling (scale-up)	Horizontal scaling (scale-out) across distributed nodes
Consistency	Strong consistency with ACID transactions	Eventual consistency, though tunable in some systems (e.g., MongoDB)
Query Language	SQL (standardized and powerful for complex queries)	Varies (e.g., MongoDB query language, Cassandra Query Language)
Transaction Support	Full ACID (Atomicity, Consistency, Isolation, Durability)	BASE model (Basically Available, Soft state, Eventually consistent); some support ACID in limited contexts
Performance	Optimized for complex queries and relationships, slower for large-scale writes	Optimized for high-throughput reads/writes, especially in distributed systems
Use Cases	Best for structured data and applications needing strong consistency (e.g., financial systems, ERP, CRM)	Best for unstructured/semi-structured data and large-scale distributed systems (e.g., big data, real-time analytics, social media)
Scaling Challenges	Difficult to shard or partition across servers	Built to shard and distribute data easily
Data Integrity	High; enforces strong data integrity through constraints and relationships	Varies; integrity is often managed at the application level
Joins and Relationships	Supports complex joins between tables and foreign key relationships	Relationships are either embedded (documents) or handled differently (e.g., graph databases)
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, DynamoDB, Redis, Neo4j

1.b	Provide strategies for optimizing data models to improve application performance. **Scheme: Explanation of any 5 optimizing models carries 2+2+2+2+2 marks each.** **Solution:** **Optimizing Data Models to Improve Application Performance** • **Denormalization** – Reduces joins by storing related data together, improving read speed. • **Indexing** – Creates indexes on frequently queried fields for faster data retrieval. • **Sharding & Partitioning** – Distributes data across multiple nodes to balance load and enhance scalability. • **Schema Optimization** – Chooses the right data model (key-value, document, column-family, graph) based on application needs. • **Caching** – Uses in-memory storage to reduce database load and speed up queries. • **Materialized Views & Batch Processing** – Precomputes and caches query results for frequently accessed data. • **Aggregate-Oriented Data Modeling** – Groups related data in NoSQL databases to minimize multiple queries. • **Compression & Serialization** – Uses efficient formats to reduce storage and improve performance. • **Replication & Load Balancing** – Distributes read/write operations across multiple database replicas for high availability. • **Concurrency Control & Transactions** – Implements appropriate consistency models for optimized performance.			
	OR			
2.a	Explain impendence mismatch problem in the context of data storage, and how does it affect application development. **Scheme: Definition + Explanation + Diagram – 2 + 5 + 3 Marks** **Solution :** The **impedance mismatch** problem arises due to the fundamental differences between **relational databases** and **in-memory data structures** used in application development. It refers to the difficulty in translating data between the **relational model (tables, rows, and columns)** and the **object-oriented programming model (objects, classes, and hierarchies)**. **Causes of Impedance Mismatch** 1. **Different Data Representations** 2. **Mapping Complexity** 3. **Joins vs. Object References** 4. **Schema Rigidity** **Impact on Application Development** The impedance mismatch problem affects application development in several ways: 1. **Increased Development Effort** 2. **Performance Overhead** 3. **Complex Queries** 4. **Data Integrity Issues** 5. **Limited Scalability** **How NoSQL Helps Solve the Impedance Mismatch** NoSQL databases (e.g., MongoDB, Cassandra) are designed to align better with object-oriented programming: • **Schema-less Design** – No rigid schemas, allowing flexible data storage. • **Aggregate-Oriented Models** – Data is stored in structured documents (JSON, BSON)	[10]	4	L2

that match application objects.

- **No Joins Needed** – Reduces the complexity of fetching related data.
- **Faster Reads/Writes** – Optimized for high-speed transactions and big data workloads.



2.b Describe the concept of schemaless databases and their benefits.

Scheme: Definition + Explanation + benefits – 2 + 5 + 3 Marks
Solution:
A **schemaless database** is a type of database that does not require a predefined structure for storing data. Unlike relational databases, which require strict schemas with predefined tables and columns, schemaless databases allow data to be stored in **flexible, dynamic formats** such as key-value pairs, documents, graphs, or wide-column structures.

Schemaless databases are commonly used in **NoSQL** systems, including **document stores (MongoDB), key-value stores (Redis), column-family stores (Cassandra), and graph databases (Neo4j)**. These databases are particularly useful for handling **semi-structured or unstructured data**, making them ideal for modern applications that require high scalability and agility.

A common theme across all the forms of NoSQL databases is that they are schemaless.
• When you want to store data in a relational database, you first must define a schema a defined structure for the database which says what tables exist, which columns exist, and what data types each column can hold.
• Before you store some data, you have to have the schema defined for it in relational database.
With a schema:
• You must figure out in advance what you need to store, but that can be hard to do.
Without a schema:
• You can easily store whatever you need.
• This allows you to easily change your data storage as you learn more about your project.
• You can easily add new things as you discover them.
• If you find you don't need some things anymore, you can just stop storing them, without worrying about losing old data as you would if you delete columns in a relational schema.
• A schema puts all rows of a table into a straightjacket, which becomes awkward if you have different kinds of data in different rows. You either end up with lots of columns that are usually null (a sparse table), or you end up with meaningless columns, like custom column 4.

Reduced Storage Overhead
Unlike relational databases that store NULL values for absent fields, schemaless databases only store **relevant data**, reducing storage costs.

Simplified Data Integration
Works well with **REST APIs and microservices** as data can be stored in a format similar to API responses (e.g., JSON).
Multi-Model Support
Some NoSQL databases support **multiple data models**, allowing flexibility in data representation.
Optimized for Distributed Systems
Designed to work across **distributed clusters**, ensuring **high availability and fault tolerance**.

Benefits of Schemaless Databases

1. Flexibility in Data Structure
2. Easy Handling of Semi-Structured Data
3. Faster Iteration and Development
4. Improved Performance and Scalability
5. Better Support for Big Data and Real-Time Analytics
6. Dynamic Indexing and Querying

Module-2

3.a Explain the concepts of Master-Slave and Peer- to-peer replication in distributed databases.
Scheme: Definition + Explanation + Diagram – 2 + 5 + 3 Marks
Solution:
Master Slave:-
● With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary.
● This master is the authoritative source for the data and is usually responsible for processing any updates to that data.
● The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master.
Peer to Peer:-
● Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master.
● Essentially, the master is still a bottleneck and a single point of failure. Peer-to-peer replication attacks these problems by not having a master.
● All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

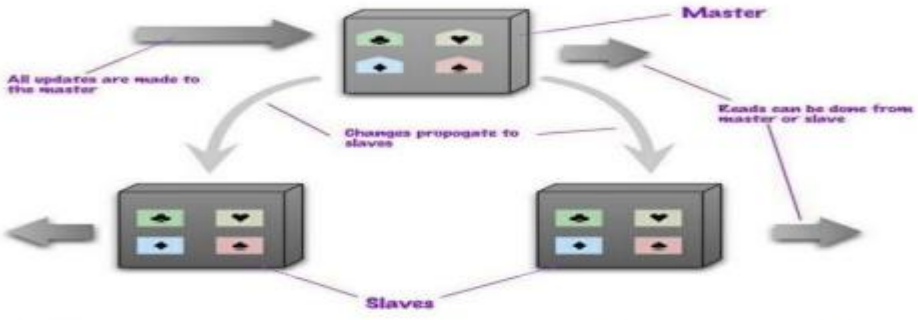


Figure 1.2. Data is replicated from master to slaves. The master services all writes; reads maycome from either master or slaves.

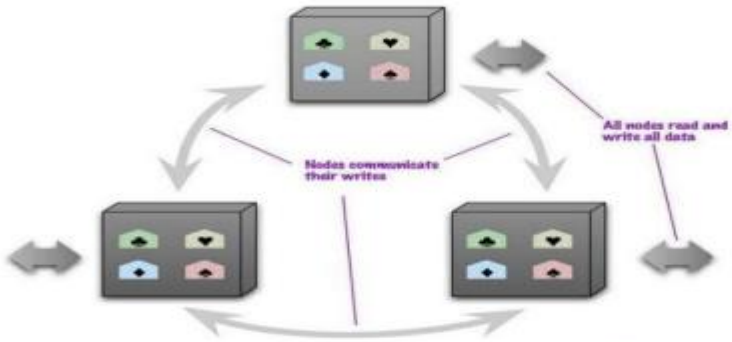


Figure 1.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

3.b	Discuss the importance of version stamp in ensuring consistency across multiple nodes in a distributed database. **Scheme: Definition + Importance + Explanation – 2 + 5+3 Marks** **Solution:** **Incremental Version Numbering** Each time data is modified, a version number is incremented. This version counter is unique to each item or record, often starting from zero. Advantages: Easy to track the order of changes, and there is no reliance on external time sources. Disadvantages: Not well-suited for distributed systems with multiple nodes updating the data simultaneously, as it can lead to version conflicts. **UUIDs (Universally Unique Identifiers)** Unique identifiers, like UUIDs, can be generated each time an item is updated, representing a new version. UUIDs use a combination of random elements and time-based factors to ensure uniqueness. Advantages: Can be generated independently on different nodes without conflict. Disadvantages: Lacks natural ordering of versions and may complicate conflict resolution. **Hash-Based Versioning** In this approach, the data's content itself is hashed (e.g., using SHA-256) to create a unique version stamp for each modification. Changes to the data will produce a different hash, indicating a new version. Advantages: Useful for ensuring data integrity, as any alteration to the data results in a new hash. Disadvantages: Hashes do not inherently indicate version order, making it harder to track the sequence of changes. **Vector Clocks** Vector clocks are a sophisticated technique used in distributed systems where each node keeps a vector of counters (one per node) for a given data item. When a node modifies the item, it increments its own counter in the vector clock. Advantages: Helps track causality and manage concurrent updates across distributed nodes. Disadvantages: Vector clocks grow in size with the number of nodes, which may lead to overhead in highly distributed environments. **Lamport Timestamps** Lamport timestamps are logical clocks that use counters instead of actual time to represent the order of events. Each process in a system maintains a counter, incrementing it with each event. This timestamp is attached to messages and helps order events in a distributed setup. Advantages: Simple to implement and suitable for distributed systems where only event order is important. Disadvantages: Only provides partial ordering of events and may not handle concurrent updates well. **Commit Hashes in Version Control Systems (e.g., Git)** Systems like Git use commit hashes (a combination of the content and parent history) to track versions. Each commit is uniquely identified, allowing for branching, merging, and comparison. Advantages: Efficient for systems requiring detailed version history with branching. Disadvantages: Not suitable for real-time versioning due to its complexity and reliance on content history.			
	OR			
4.a	Discuss the challenges associated with achieving update consistency in a distributed database system. **Scheme : Definition + challenges + explanation with example– 2+4+4 Marks** **Solution :**	[5]	4	L2

We'll begin by considering updating a telephone number. Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. Implausibly, they bothhave update access, so they both go in at the same time to update the number. To make the example interesting, we'll assume they update it slightly differently, because each uses a slightly different format. This issue is called a **write-write conflict**: two people updating the same data item at the same time.

When the writes reach the server, the server will **serialize** them—decide to apply one, then the other. Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's. Without any concurrency control, Martin's update would be applied and immediately overwritten by Pramod's. In this case Martin's is a **lost update**. Here the lost update is not a big problem, but often it is. We see this as a failure of consistency because Pramod's update was based on the state before Martin's update, yet was applied after it.

Approaches for maintaining consistency in the face of concurrency are often described as pessimistic or optimistic. A pessimistic approach works by preventing conflicts from occurring; an optimistic approach lets conflicts occur but detects them and takes action to sort them out. For update conflicts, the most common pessimistic approach is to have write locks, so that in order to change a value you need to acquire a lock, and the system ensures that only one client can get a lock at a time.

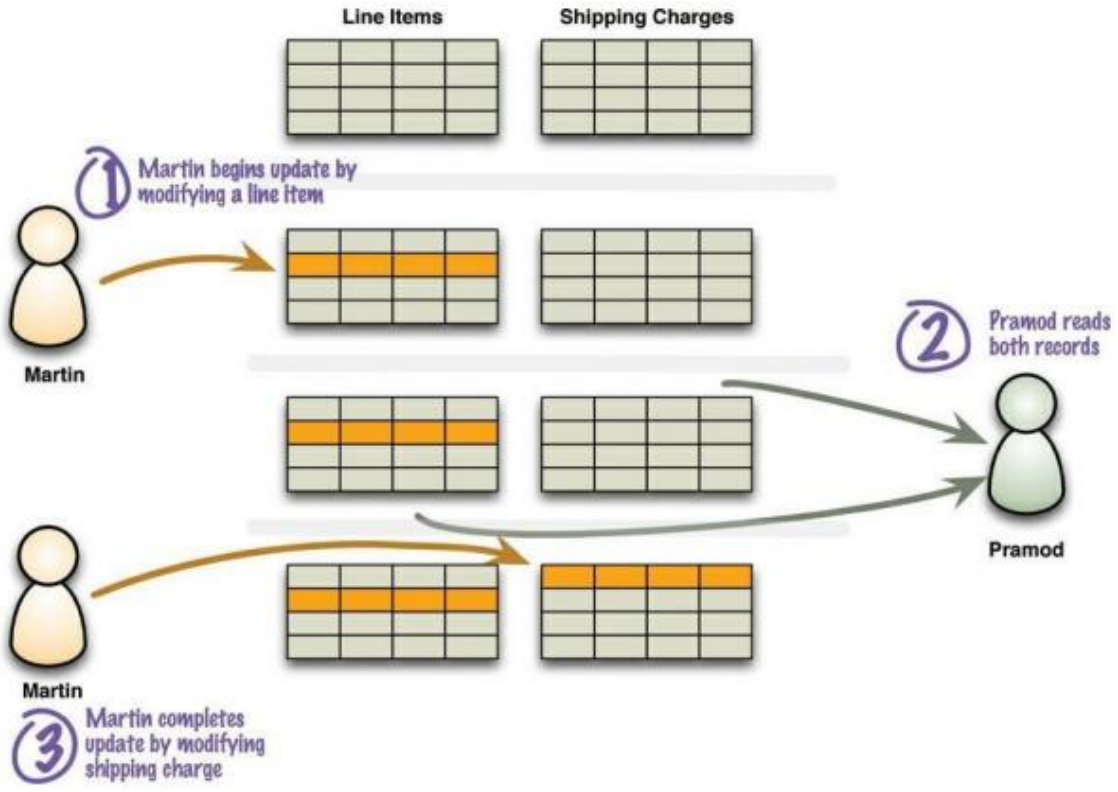
So Martin and Pramod would both attempt to acquire the write lock, but only Martin (the first one) would succeed. Pramod would then see the result of Martin's write before deciding whether to make his own update.

A common optimistic approach is a conditional update where any client that does an update tests the value just before updating it to see if it's changed since his last read. In this case, Martin's update would succeed but Pramod's would fail. The error would let Pramod know that he should look at the value again and decide whether to attempt a further update.

Both the pessimistic and optimistic approaches that we've just described rely on a consistent serialization of the updates. With a single server, this is obvious it has to choose one, then the other. But if there's more than one server, such as with peer-to-peer replication, then two nodes might apply the updates in a different order, resulting in a different value for the telephone number on each peer.

Often, when people talk about concurrency in distributed systems, they talk about sequential consistency—ensuring that all nodes apply operations in the same order.

There is another optimistic way to handle a write-write conflict—save both updates and record that they are in conflict. This approach is familiar to many programmers from version control systems, particularly distributed version control systems that by their nature will often have conflicting commits. The next step again follows from version control: You have to merge the two updates somehow. Maybe you show both values to the user and ask them to sort it out—this is what happens ifyou update the same contact on your phone and your computer. Alternatively, the computer may be able to perform the merge itself; if it was a phone formatting issue, it may be able to realize that and apply the new number with the standard format. Any automated merge of write-write conflicts is highly domain-specific and needs to be programmed for each particular case.

4.b	Discuss read consistency-in distributed databases, considering factors such as staleness and isolation levels. **Scheme : Definition + explanation + Diagram – 2+4+4 Marks** **Solution :** In distributed databases, **read consistency** ensures that users receive accurate and expected data despite multiple copies being stored across different nodes. Read consistency is influenced by factors like **staleness, isolation levels, and consistency models** used by the database system. **Factors Affecting Read Consistency** **a) Staleness** Staleness refers to the delay between **data updates** on one node and its availability on other nodes. Causes of staleness: • **Replication Lag**: In eventually consistent systems, updates take time to propagate. • **Network Latency**: Delays in synchronization between distributed nodes. • **Read Preference**: Reading from secondary replicas may return outdated data. ◆ **Example:** A user updates their profile picture, but another user still sees the old picture due to delayed synchronization across replicas. **b) Isolation Levels** Isolation levels determine **how concurrent transactions** interact with each other. Lower isolation may lead to inconsistent reads. The common **isolation levels** include: 1. **Read Uncommitted** 2. **Read Committed** 3. **Repeatable Read** 4. **Serializable**  **Figure 2.1. A read-write conflict in logical consistency**	[5]		

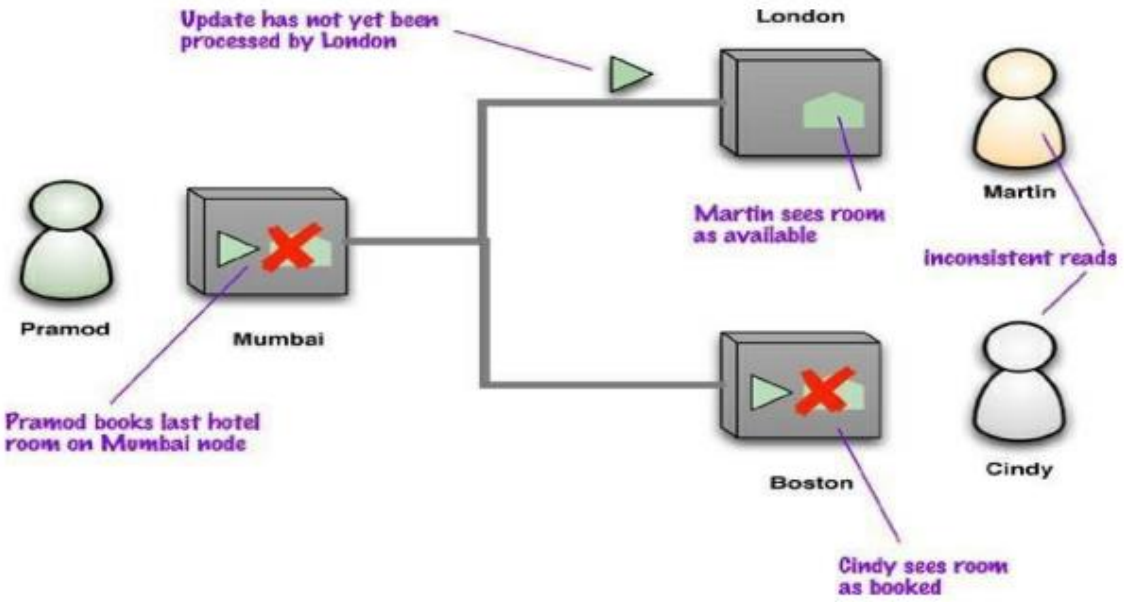


Figure 1.2. An example of replication inconsistency

Module-3

5.a Provide an example of composing Map-Reduce calculations to process and Analyze data. [10] 3 L2

Scheme : Definition + explanation + Diagram – 2+4+4 Marks

Solution :

The map-reduce approach is a way of thinking about concurrent processing that trades off flexibility in how you structure your computation for a relatively straightforward model for parallelizing the computation over a cluster. Since it's a tradeoff, there are constraints on what you can do in your calculations. Within a map task, you can only operate on a single aggregate. Within a reduce task, you can only operate on a single key. This means you have to think differently about structuring your programs so they work well within these constraints.

One simple limitation is that you have to structure your calculations around operations that fit in well with the notion of a reduce operation. A good example of this is calculating averages. Let's consider the kind of orders we've been looking at so far; suppose we want to know the average ordered quantity of each product. An important property of averages is that they are not comparable.

that is, if I take two groups of orders, I can't combine their averages alone. Instead, I need to take total amount and the count of orders from each group, combine those, and then calculate the average from the combined sum and count (see Figure 1.6).

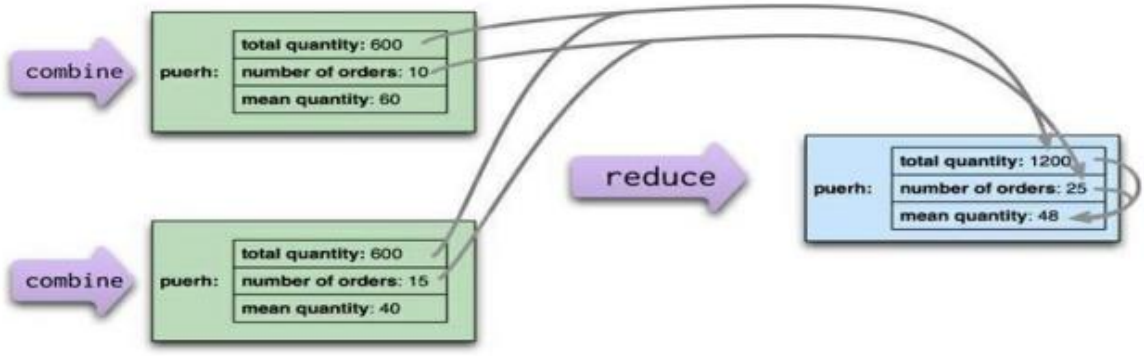


Figure 1.6. When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.

This notion of looking for calculations that reduce neatly also affects how we do counts. To make a count, the mapping function will emit count fields with a value of 1, which can be summed to get a total count (see Figure 1.7).

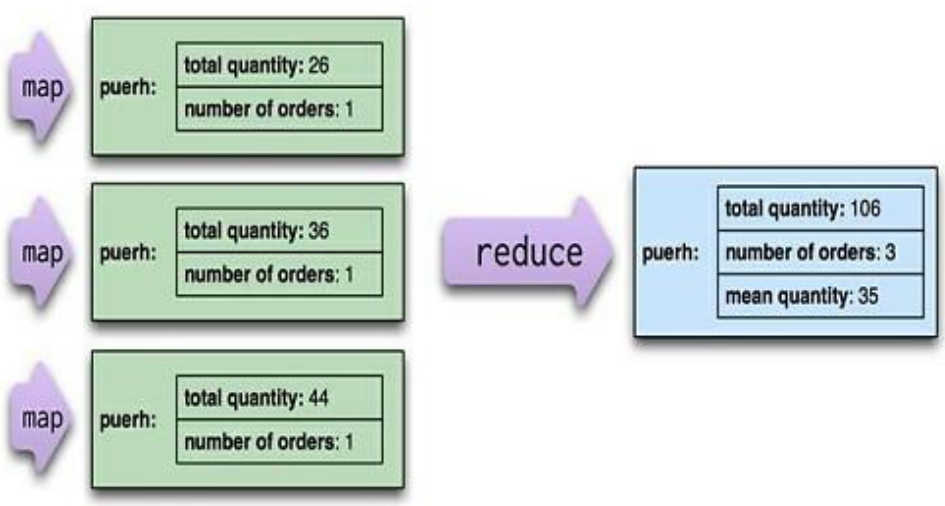


Figure 1.7. When making a count, each map emits 1, which can be summed to get a total.

5.b Discuss the scalability characteristics of key-value databases.

Scheme : Definition + explanation of each – 2+8 Marks

Solution :

Key-value databases are known for their high scalability, making them ideal for large-scale distributed applications. They achieve scalability by using techniques such as sharding, replication, and eventual consistency.

Below are the key scalability characteristics of key-value stores:

1. Horizontal Scalability (Scale-Out)

- Key-value databases can scale horizontally by distributing data across multiple nodes.
- Unlike relational databases that rely on vertical scaling (adding more CPU, RAM, or storage to a single machine), key-value stores support horizontal scaling (adding more machines to handle the load).
- Example: Amazon DynamoDB and Apache Cassandra can dynamically add more nodes to handle increased demand.

2. Partitioning (Sharding) for Load Distribution

- Data is distributed across multiple shards (partitions) using techniques like:
  - Hash-based partitioning: Distributes keys based on a hash function (e.g., hash(key) % number_of_shards).
  - Range-based partitioning: Assigns key ranges to different servers (e.g., keys A–M on one node, N–Z on another).
- Sharding allows even distribution of load, preventing bottlenecks on a single server.

3. Replication for High Availability

- Key-value databases use replication to create multiple copies of data across nodes, ensuring high availability.
- Types of replication:
  - Master-Slave Replication: A primary node handles writes, and secondary nodes handle reads (used in Redis).
  - Peer-to-Peer (Multi-Master) Replication: Any node can handle reads and writes, improving scalability (used in Cassandra and DynamoDB).

4. Eventual Consistency for Performance Optimization

- Many key-value databases use eventual consistency to improve scalability.
- Instead of enforcing strong consistency (which slows down operations), updates propagate asynchronously across nodes.
- Example: Amazon DynamoDB and Apache Cassandra allow temporary data inconsistencies, improving speed.

5. Low-Latency Reads and Writes

- Key-value databases are optimized for constant-time (O(1)) lookup of values using unique keys.
- They avoid expensive SQL queries, complex joins, and transactions, making them ideal for real-time applications.
- Example:
  - Redis: Uses in-memory storage for sub-millisecond response times.
  - DynamoDB: Provides single-digit millisecond latency for global-scale applications.

◆ Use Cases: Web caching, real-time analytics, and gaming leaderboards.

6. Elastic Scaling for Cloud-Native Applications

- Cloud-based key-value stores (e.g., AWS DynamoDB, Google Cloud Bigtable) offer on-demand auto-scaling, adjusting resources based on traffic patterns.
- No need for manual intervention—cloud services automatically handle node allocation.

7. Schema-Free Data Model for Dynamic Scaling

- No strict schema requirements mean data structure can evolve dynamically without downtime.
- This flexibility supports heterogeneous data types, making it suitable for IoT, logs, and session management.

◆ Example:

- A product catalog where different items have different attributes (e.g., "Laptop" may have RAM, while "Book" has an author field).

8. Distributed Query Execution with Parallel Processing

- Some key-value stores (e.g., Cassandra, Aerospike) distribute queries across nodes to improve read performance.
- Parallel query execution reduces response time for large-scale datasets.

◆ Use Case: Large-scale recommendation engines and real-time fraud detection.

OR

6.a Discuss key features of key value stores and their advantages. [10]

Scheme : Features + explanation of each + advantages – 8+2 Marks

Solution :

Consistency

Consistency is applicable only for operations on a single key, since these operations are either a get, put, or delete on a single key. Optimistic writes can be performed, but are very expensive to implement, because a change in value cannot be determined by the data store.

In distributed key-value store implementations like Riak, the eventually consistent (p. 50) model of consistency is implemented. Since the value may have already been replicated to other nodes, Riak has two ways of resolving update conflicts: either the newest write wins and older writes loose, or both (all) values are returned allowing the client to resolve the conflict.

In Riak, these options can be set up during the bucket creation. Buckets are just a way to namespace keys so that key collisions can be reduced—for example, all customer keys may reside in the customer bucket. When creating a bucket, default values for consistency can be provided, for example that a write is considered good only when the data is consistent across all the nodes where the data is stored.

```
Bucket bucket = connection
.createBucket(bucketName)
.withRetrier(attempts(3))
.allowSiblings(siblingsAllowed)
.nVal(numberOfReplicasOfTheData)
.w(numberOfNodesToRespondToWrite)
.r(numberOfNodesToRespondToRead)
.execute();
```

Transactions

Different products of the key-value store kind have different specifications of transactions. Generally speaking, there are no guarantees on the writes. Many data stores do implement transactions in different ways. Riak uses the concept of quorum (“Quorums,” p. 57) implemented by using the Wvalue

—replication factor—during the write API call.

Assume we have a Riak cluster with a replication factor of 5 and we supply the Wvalue of 3. When writing, the write is reported as successful only when it is written and reported as a success on at least three of the nodes. This allows Riak to have write tolerance; in our example, with Nequal to 5

and with a Wvalue of 3, the cluster can tolerate N - W = 2nodes being down for write operations, though we would still have lost some data on those nodes for read.

Query Features

All key-value stores can query by the key—and that’s about it. If you have requirements to query by using some attribute of the value column, it’s not possible to use the database: Your application needs to read the value to figure out if the attribute meets the conditions.

Query by key also has an interesting side effect. What if we don’t know the key, especially during ad-hoc querying during debugging? Most of the data stores will not give you a list of all the primary keys; even if they did, retrieving lists of keys and then querying for the value would be very cumbersome. Some key-value databases get around this by providing the ability to search inside the value, such as Riak Search that allows you to query the data just like you would query it using Lucene indexes.

While using key-value stores, lots of thought has to be given to the design of the key. Can the key be generated using some algorithm? Can the key be provided by the user (user ID, email, etc.)? Or derived from timestamps or other data that can be derived outside of the database?

These query characteristics make key-value stores likely candidates for storing session data (with the session ID as the key), shopping cart data, user profiles, and so on. The expiry_secs property can be used to expire keys after a certain time interval, especially for session/shopping cart objects.

Bucket bucket = getBucket(bucketName);

IRiakObject riakObject = bucket.store(key, value).execute();

When writing to the Riak bucket using the storeAPI, the object is stored for the key provided.

Similarly, we can get the value stored for the key using the fetchAPI. Bucket bucket = getBucket(bucketName);

IRiakObject riakObject

=

bucket.fetch(key).execute(); byte[]

bytes

=

riakObject.getValue();

String value = new String(bytes);

Riak provides an HTTP-based interface, so that all operations can be performed from the web browser or on the command line using curl. Let’s save this data to Riak:

{

"lastVisit":132 4669989288,

"user":{

"customerId ":"91cfdf5b cb7c", "name":"bu yer", "countryCo de":"US",

"tzOffset":0

}

}

Use the curlcommand to POSTthe data, storing the data in the sessionbucket with the key of a7e618d9db25 (we have to providethis key):

curl -v -X POST -d '

{ "lastVisit":1324669989288,

"user":{"customerId":"91cfdf5bcb7c", "name":"buyer",

"countryCo de":"US",

"tzOffset":0

}

}'

-H "Content-Type: application/json" http://localhost:8098/buckets/session/keys/a7e618d9db25

The data for the key a7e618d9db25can be fetched by using the curlcommand:

	curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25 **Structure of Data** Key-value databases don't care what is stored in the value part of the key-value pair. The value can be a blob, text, JSON, XML, and so on. In Riak, we can use the Content-Type in the POST request to specify the data type. **Scaling** Many key-value stores scale by using sharding ("Sharding," p. 38). With sharding, the value of the key determines on which node the key is stored. Let's assume we are sharding by the first character of the key; if the key is f4b19d79587d, which starts with an f, it will be sent to different node than the key ad9c7a396542. This kind of sharding setup can increase performance as more nodes are added to the cluster. Sharding also introduces some problems. If the node used to store f goes down, the data stored on that node becomes unavailable, nor can new data be written with keys that start with f. Data stores such as Riak allow you to control the aspects of the CAP Theorem ("The CAP Theorem," p. 53): N (number of nodes to store the key-value replicas), R(number of nodes that have to have the data being fetched before the read is considered successful), and W (the number of nodes the write has to be written to before it is considered successful).			
6.b	Describe the basic structure of data in a key value databases. **Scheme- Definition + Explanation- 5+5** **Solution-** •Hash Table Structure: At its core, a key-value store is a simple hash table where all data access occurs via the primary key. •Key-Value Pairs: The fundamental unit of data is a key-value pair. The key acts as a unique identifier, similar to an ID in a relational database, while the value stores the actual data. The application is responsible for understanding the structure and format of the data stored in the value. •Keys and Values: ◦Key: The key serves as the primary means of accessing the stored data. It is typically a string or a unique identifier. ◦Value: The value can be a variety of data types, including blobs, text, JSON, or XML. The database does not interpret the value; it simply stores and retrieves it. •Buckets for Organization: Keys can be organized into buckets, which serve as flat namespaces. Buckets help segment data and reduce the chance of key conflicts. For example, all customer keys might reside in a "customer" bucket. •Flexibility in Data Storage: Key-value stores do not enforce a schema on the value, providing flexibility in how data is stored. The value can be a simple data structure or a complex aggregate.	[10]		
	MODULE-4			
7.a	Explain fundamental principles of document database. **Scheme- Definition + Explanation- 4+6** **Solution-** The fundamental principles of document databases are centered around the concept of storing and retrieving self-describing, hierarchical data structures known as documents. •Document-Oriented Storage: Document databases store data in the form of documents, which can be represented in formats like JSON or XML. These documents are stored in the value part of a key-value store. •Flexible Schema: Unlike relational databases, document databases do not require a fixed schema. Documents within the same collection can have different attributes. If an attribute is absent in a document, it is considered unset or irrelevant, rather than being assigned an empty or null value. •Hierarchical Structure: Documents can have a hierarchical tree structure consisting of maps, collections, and scalar values. They can also embed child documents as sub-objects. •Unique Identification: Similar to ROWID in Oracle, MongoDB uses the _id field to uniquely identify each document. Users can assign this _id, as long as it remains unique. •Collections: MongoDB instances consist of databases, each containing multiple collections. These collections are analogous to tables in relational database management systems (RDBMS). When storing a document, a database and collection must be selected. •Querying: Document databases allow querying of data within a document without needing to	[10]		

	retrieve the entire document using its key. MongoDB uses a JSON-based query language with constructs like $query (for the where clause), $orderby (for sorting), and $explain (to show the execution plan). •Scaling: Document databases support horizontal scaling, which involves adding nodes or changing data storage without migrating to a larger system. Scaling for read-heavy loads can be achieved by adding more read slaves. For scaling writes, data can be sharded across multiple MongoDB nodes. •Availability and Consistency: Document databases often use master-slave setups to replicate data across multiple nodes, enhancing availability. MongoDB implements replication using replica sets. Consistency can be configured by specifying the number of slaves a write operation must be propagated to before being considered successful. •Transactions: Traditional RDBMS transactions are generally not available in NoSQL solutions. However, some document databases like RavenDB support transactions across multiple operations. Atomic transactions are available at the single-document level.			
7.b	Discuss the importance of SEO*(Search Engine Optimization) in the context of blogging platforms. **Scheme- Definition+explanation= 4+6** **Solution-** A key component of effective blogging is search engine optimization (SEO), which raises a blog's exposure and accessibility to a wider audience. Bloggers may improve user experience, build authority in their field, and draw in more organic traffic by optimizing their content for search engines. Enhanced Organic Traffic: By improving your blog's ranking in search engine results pages (SERPs), readers are more likely to see your material when they are looking for related topics. Increased Credibility and Trust: Users tend to view blogs that rank highly in search results as more reliable and trustworthy, which can increase user engagement and loyalty. Better User Experience: SEO entails improving a number of aspects of your blog, including content organization, mobile responsiveness, and site speed, all of which enhance the reading experience for your audience. Competitive advantage: By using efficient SEO techniques, you can differentiate your blog in the crowded blogosphere and attract and hold on to a specific audience.	[10]		
	OR			
8.a	Provide examples of common query operations performed on document databases and explain their significance. **Scheme- Definition + Explanation- 4+6** **Solution-** Document databases provide different query features. CouchDB allows you to query via views—complex queries on documents which can be either materialized or dynamic. With CouchDB, if you need to aggregate the number of reviews for a product as well as the average rating, you could add a view implemented via map-reduce to return the count of reviews and the average of their ratings. When there are many requests, you don’t want to compute the count and average for every request; instead you can add a materialized view that precomputes the values and stores the results in the database. These materialized views are updated when queried, if any data was changed sincethe last update. One of the good features of document databases, as compared to key-value stores, is that we can query the data inside the document without having to retrieve the whole document by its key and then introspect the document. This feature brings these databases closer to the RDBMS query model.	[10]		

	MongoDB has a query language which is expressed via JSON and has constructs such as $query for the where clause, $orderby for sorting the data, or $explain to show the execution plan of the query. There are many more constructs like these that can be combined to create a MongoDB query. Let's look at certain queries that we can do against MongoDB. Suppose we want to return all the documents in an order collection (all rows in the order table). The SQL for this would be: SELECT * FROM order The equivalent query in Mongo shell would be: db.order.find() Selecting the orders for a single customerIdof 883c2c5b4e5bwould be: SELECT * FROM order WHERE customerId = "883c2c5b4e5b" The equivalent query in Mongo to get all orders for a single customerIdof 883c2c5b4e5b: db.order.find({"customerId":"883c2c5b4e5b"}) Similarly, selecting orderIdand orderDatefor one customer in SQL would be: SELECT orderId,orderDate FROM order WHERE customerId = "883c2c5b4e5b" and the equivalent in Mongo would be: db.order.find({customerId:"883c2c5b4e5b"},{orderId:1,orderDate:1}) Similarly, queries to count, sum, and so on are all available. Since the documents are aggregated objects, it is really easy to query for documents that have to be matched using the fields with child objects. Let's say we want to query for all the orders where one of the items ordered has a name like Refactoring. The SQL for this requirement would be: SELECT * FROM customerOrder, orderItem, product WHERE customerOrder.orderId = orderItem.customerOrderId AND orderItem.productId = product.productId AND product.name LIKE '%Refactoring%' and the equivalent Mongo query would be: db.orders.find({"items.product.name":/Refactoring/})			
8.b	Explain the importance of event logging in web applications with examples. **Scheme- Explanation + Example- 7+3** **Solution-** Event Logging Applications have different event logging needs; within the enterprise, there are many different applications that want to log events. Document databases can store all these different types of events and can act as a central data store for event storage. This is especially true when the type of data being captured by the events keeps changing. Events can be sharded by the name of the application where the event originated or by the type of event such as order_processed or customer_logged. •Centralized Event Storage: Document databases can serve as a central repository for storing various types of events from different applications. This allows for a unified view of application activities. •Flexible Schema for Diverse Events: Given that event data can vary significantly across applications, document databases' schema flexibility is highly beneficial. This allows for capturing different event attributes without needing to alter a predefined schema. •Real-Time Analytics: Document databases can store data for real-time analytics. Parts of the document can be updated easily, making it simple to store metrics like page views or unique visitors. New metrics can be added without schema changes. •Sharding for Scalability: Events can be sharded by application name or event type (e.g., order_processed, customer_logged). This enables horizontal scaling to handle large volumes of event data. Examples: ◦Application Event Logging: Different applications can log events such as user logins, order placements, and system errors into a document database. ◦Web Analytics: Capturing and storing user interactions such as page views, clicks, and form submissions for analyzing user behavior.	[10]		

MODULE-5

9.a Discuss the key features of graph databases that make suitable for handling connected data. [10]

Scheme- Features + Explanation- 4+6

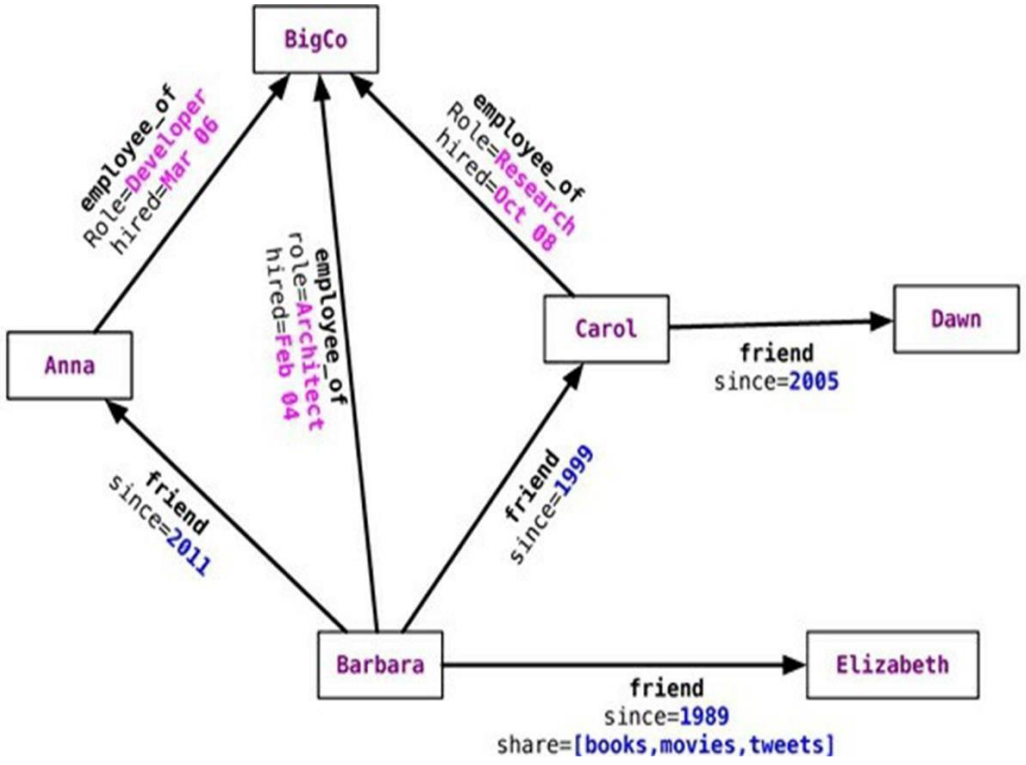
Solution-

Features

Consistency

Since graph databases are operating on connected nodes, most graph database solutions usually do not support distributing the nodes on different servers. There are some solutions, however, that support node distribution across a cluster of servers, such as Infinite Graph. Within a single server, data is always consistent, especially in Neo4J which is fully ACID- compliant. When running Neo4J in a cluster, a write to the master is eventually synchronized to the slaves, while slaves are always available for read. Writes to slaves are allowed and are immediately synchronized to the master; other slaves will not be synchronized immediately, though—they will have to wait for the data to propagate from
the master.

Graph databases ensure consistency through transactions. They do not allow dangling relationships: The start node and end node always have to exist, and nodes can only be deleted if they don't have any relationships attached to them.



Transactions

Neo4J is ACID-compliant. Before changing any nodes or adding any relationships to existing nodes, we have to start a transaction. Without wrapping operations in transactions, we will get a NotInTransactionException. Read operations can be done without initiating a transaction.

Transaction transaction = database.beginTx(); try {
Node node = database.createNode(); node.setProperty("name", "NoSQL Distilled"); node.setProperty("published", "2012"); transaction.success();
} finally { transaction.finish();
}

In the above code, we started a transaction on the database, then created a node and set properties on it. We marked the transaction as success and finally completed it by finish. A transaction has to be marked as success, otherwise Neo4J assumes that it was a failure and rolls it back when finish is issued. Setting success without issuing finish also does not commit the data to the database. This way of managing transactions has to be remembered when developing, as it differs from the standard way of doing transactions in an RDBMS.

Availability

Neo4J, as of version 1.8, achieves high availability by providing for replicated slaves. These slaves can also handle writes: When they are written to, they synchronize the write to the current master, and the write is committed first at the master and then at the slave. Other slaves will eventually get the update. Other graph databases, such as Infinite Graph and FlockDB, provide for distributed storage of the nodes.

Query Features

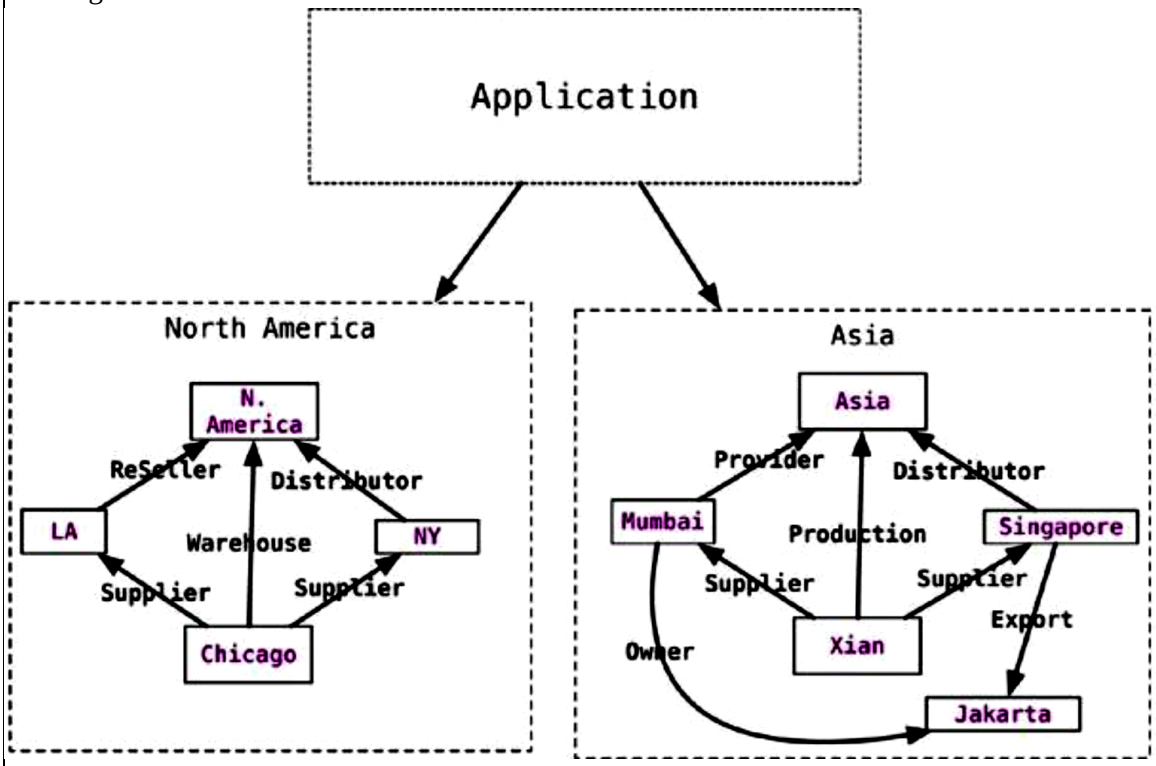
Graph databases are supported by query languages such as Gremlin [Gremlin]. Gremlin is a domain- specific language for traversing graphs; it can traverse all graph databases that implement the Blueprints [Blueprints] property graph. Neo4J also has the Cypher [Cypher] query language for querying the graph. Outside these query languages, Neo4J allows you to query the graph for properties of the nodes, traverse the graph, or navigate the nodes relationships using language bindings.

Properties of a node can be indexed using the indexing service. Similarly, properties of relationships or edges can be indexed, so a node or edge can be found by the value. Indexes should be queried to find the starting node to begin a traversal. Let's look at searching for the node using node indexing.

Scaling

In NoSQL databases, one of the commonly used scaling techniques is sharding, where data is split and distributed across different servers. With graph databases, sharding is difficult, as graph databases are not aggregate-oriented but relationship-oriented. Since any given node can be related to any other node, storing related nodes on the same server is better for graph traversal. Traversing a graph when the nodes are on different machines is not good for performance. Knowing this limitation of the graph databases, we can still scale them using some common techniques described by Jim Webber [Webber Neo4J Scaling].

Generally speaking, there are three ways to scale graph databases. Since machines now can come with lots of RAM, we can add enough RAM to the server so that the working set of nodes and relationships is held entirely in memory. This technique is only helpful if the dataset that we are working with will fit in a realistic amount of RAM.



9.b	Identify scenarios where using a graph database may not be appropriate what are the limitations. **Scheme- Definition + Explanation- 4+6** **Solution-**	[10]		

In some situations, graph databases may not appropriate. When you want to update all or a subset of entities—for example, in an analytics solution where all entities may need to be updated with a changed property—graph databases may not be optimal since changing a property on all the nodes is not a straightforward operation. Even if the data model works for the problem domain, some databases may be unable to handle lots of data, especially in global graph operations (those involving the whole graph) Computations.

•Updating Large Subsets of Entities: Graph databases may not be optimal for analytics solutions where a changed property needs to be updated across a large subset or all entities. Changing a property on all the nodes is not a straightforward operation.

•Global Graph Operations: Some graph databases may struggle with global graph operations, especially when dealing with large datasets. Some databases may be unable to handle lots of data, especially in global graph operations.

•Sharding Challenges: Sharding is difficult in graph databases because they are relationship-oriented rather than aggregate-oriented. Traversing a graph when nodes are on different machines impacts performance.

•When updates are applied to all or a subset of entities: Graph databases may not be optimal.

OR

10.a Describe how transactions are handled in a graph database. What are the ACID properties are implemented. [10]

Scheme- Transaction + ACID properties- 5+5

Solution-

Neo4J is ACID-compliant. Before changing any nodes or adding any relationships to existing nodes, we have to start a transaction. Without wrapping operations in transactions, we will get a NotInTransactionException. Read operations can be done without initiating a transaction.

```
Transaction transaction = database.beginTx(); try {
Node node = database.createNode(); node.setProperty("name", "NoSQL Distilled");
node.setProperty("published", "2012"); transaction.success();
} finally { transaction.finish();
}
```

In the above code, we started a transaction on the database, then created a node and set properties on it. We marked the transaction as success and finally completed it by finish. A transaction has to be marked as success, otherwise Neo4J assumes that it was a failure and rolls it back when finish is issued. Setting success without issuing finish also does not commit the data to the database. This way of managing transactions has to be remembered when developing, as it differs from the standard way of doing transactions in an RDBMS.

ACID Properties:

◦Atomicity: All operations within a transaction are treated as a single unit; either all changes are applied, or none are.

◦Consistency: Transactions ensure that the database remains in a consistent state, preventing dangling relationships. The start node and end node always have to exist, and nodes can only be deleted if they don't have any relationships attached to them.

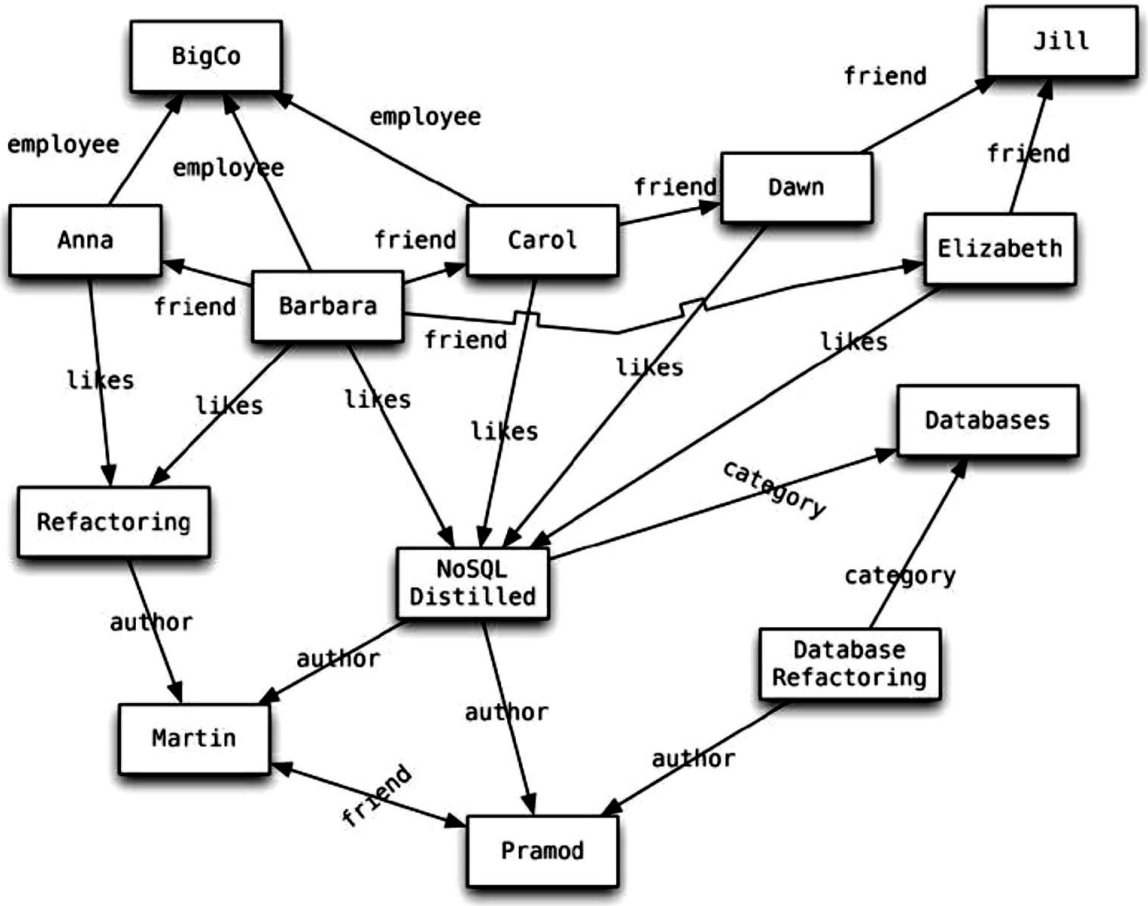
◦Isolation: Concurrent transactions are isolated from each other, preventing interference and ensuring that each transaction operates as if it were the only one running.

◦Durability: Once a transaction is committed, the changes are permanent and will survive even system failures.

10. b Provide examples of complex queries that can be efficiency executed in graph database. [10]

Scheme- Queries + example= 7+3

Solution-



If we have the graph shown in Figure 5.1, we can index the nodes as they are added to the database, or we can index all the nodes later by iterating over them. We first need to create an index for the nodes using the IndexManager.

Index<Node> nodeIndex = graphDb.index().forNodes("nodes");

We are indexing the nodes for the name property. Neo4J uses Lucene [Lucene] as its indexing service. We will see later that we can also use the full-text search capability of Lucene. When new nodes are created, they can be added to the index.

Transaction transaction = graphDb.beginTx(); try {

Index<Node> nodeIndex = graphDb.index().forNodes("nodes"); nodeIndex.add(martin, "name", martin.getProperty("name")); nodeIndex.add(pramod, "name", pramod.getProperty("name")); transaction.success();

} finally { transaction.finish();

}

Adding nodes to the index is done inside the context of a transaction. Once the nodes are indexed, we can search them using the indexed property. If we search for the node with the name of Barbara, we would query the index for the property of name to have a value of Barbara.

Node node = nodeIndex.get("name", "Barbara").getSingle();

We get the node whose name is Martin; given the node, we can get all its relationships.

Node martin = nodeIndex.get("name", "Martin").getSingle(); allRelationships = martin.getRelationships();

We can get both INCOMING or OUTGOING relationships. incomingRelations = martin.getRelationships(Direction.INCOMING);

We can also apply directional filters on the queries when querying for a relationship. With the graph in Figure 5.1, if we want to find all people who like NoSQL Distilled, we can find the NoSQL Distilled node and then get its relationships with Direction.INCOMING. At this point we can also add the type of relationship to the query filter, since we are looking only for nodes that LIKE NoSQL Distilled.

Node nosqlDistilled = nodeIndex.get("name", "NoSQL Distilled").getSingle();

relationships = nosqlDistilled.getRelationships(INCOMING, LIKES); for (Relationship relationship : relationships) { likesNoSQLDistilled.add(relationship.getStartNode());
}
Finding nodes and their immediate relations is easy, but this can also be achieved in RDBMS databases. Graph databases are really powerful when you want to traverse the graphs at any depth and specify a starting node for the traversal. This is especially useful when you are trying to find nodes that are related to the starting node at more than one level down. As the depth of the graph increases, it makes more sense to traverse the relationships by using a Traverser where you can specify that you are looking for INCOMING, OUTGOING, or BOTH types of relationships. You can also make the traverser go top-down or sideways on the graph by using Order values of BREADTH_FIRST or DEPTH_FIRST. The traversal has to start at some node—in this example, we try to find all the nodes at any
depth that are related as a FRIEND with Barbara:
Node barbara = nodeIndex.get("name", "Barbara").getSingle(); Traverser friendsTraverser = barbara.traverse(Order.BREADTH_FIRST,
StopEvaluator.END_OF_GRAPH, ReturnableEvaluator.ALL_BUT_START_NODE, EdgeType.FRIEND, Direction.OUTGOING);
The friendsTraverser provides us a way to find all the nodes that are related to Barbara where the relationship type is FRIEND. The nodes can be at any depth—friend of a friend at any level—allowing you to explore tree structures.

One of the good features of graph databases is finding paths between two nodes— determining if there are multiple paths, finding all of the paths or the shortest path. In the graph in Figure 11.1, we know that Barbara is connected to Jill by two distinct paths; to find all these paths and the distance between Barbara and Jill along those different paths, we can use
Node barbara = nodeIndex.get("name", "Barbara").getSingle(); Node jill = nodeIndex.get("name", "Jill").getSingle(); PathFinder<Path> finder = GraphAlgoFactory.allPaths(Traversal.expanderForTypes(FRIEND,Direction.OUTGOING)
,MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
This feature is used in social networks to show relationships between any two nodes. To find all the paths and the distance between the nodes for each path, we first get a list of distinct paths between the two nodes. The length of each path is the number of hops on the graph needed to reach the destination node from the start node. Often, you need to get the shortest path between two nodes; of the two paths from Barbara to Jill, the shortest path can be found by using
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(Traversal.expanderForTypes(FRIEND, Direction.OUTGOING), MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
Many other graph algorithms can be applied to the graph at hand, such as Dijkstra's algorithm [Dijkstra's] for finding the shortest or cheapest path between nodes.
START beginingNode = (beginning node specification) MATCH (relationship, pattern matches) WHERE (filtering condition: on data in nodes and relationships) RETURN (What to return: nodes, relationships, properties) ORDER BY (properties to order by)
SKIP (nodes to skip from top) LIMIT (limit results)

Neo4J also provides the Cypher query language to query the graph. Cypher needs a node to START the query. The start node can be identified by its node ID, a list of node IDs, or index lookups. Cypher uses the MATCH keyword for matching patterns in relationships; the WHERE keyword filters the properties on a node or relationship. The RETURN keyword specifies what gets returned by the query—nodes, relationships, or fields on the nodes or relationships.
Cypher also provides methods to ORDER, AGGREGATE, SKIP, and LIMIT the data. In Figure 5.2, we find all nodes connected to Barbara, either incoming or outgoing, by using the --.
START barbara = node:nodeIndex(name = "Barbara") MATCH (barbara)-- (connected_node) RETURN connected_node
When interested in directional significance, we can use MATCH (barbara)<--(connected_node)

	for incoming relationships or MATCH (barbara)-->(connected_node) for outgoing relationships. Match can also be done on specific relationships using the :RELATIONSHIP_TYPE convention and returning the required fields or nodes. START barbara = node:nodeIndex(name = "Barbara") MATCH (barbara)-[:FRIEND]->(friend_node) RETURN friend_node.name,friend_node.location We start with Barbara, find all outgoing relationships with the type of FRIEND, and return the friends' names. The relationship type query only works for the depth of one level; we can make it work for greater depths and find out the depth of each of the result nodes. Click here to view code image START barbara=node:nodeIndex(name = "Barbara") MATCH path = barbara- [:FRIEND*1..3]->end_node RETURN barbara.name,end_node.name, length(path) Similarly, we can query for relationships where a particular relationship property exists. We can also filter on the properties of relationships and query if a property exists or not. START barbara = node:nodeIndex(name = "Barbara") MATCH (barbara)-[relation]->(related_node)			

Faculty Signature CCI Signature HOD Signature