USN ☐☐☐☐☐☐☐☐☐☐

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A++ GRADE BY NAAC

Internal Assessment Test 3 – Dec 2024

Sub:	**NOSQL DATABASE**			Sub Code:	**21CS745**	Branch:	**ISE**
Date:	**14/12/2024**	Duration:	**90 min**	Max Marks: **50**	Sem/Sec:	**VII/ A, B & C**	**OBE**

Answer any FIVE FULL Questions MARKS CO RBT

1. Explain consistency and availability in MongoDB with a neat diagram for configuration of replica sets. [10] 3 L2

Scheme: Definition + Explanation + Example + Diagram – 2+3+2+3 Marks
Solution:

Consistency and Availability in MongoDB

MongoDB, like other distributed databases, follows the principles of **CAP theorem**, which states that a distributed system can provide only two out of the three guarantees: **Consistency**, **Availability**, and **Partition Tolerance**.
In the case of MongoDB:

- **Consistency**: Ensures that once data is written to a replica set, all subsequent reads will return the most recent write. This means that after a write operation, the system will ensure that all nodes in the replica set are consistent before responding to a read request.
- **Availability**: Refers to the system's ability to always respond to requests, even if some of the nodes in the replica set are unavailable. MongoDB achieves availability by ensuring that at least one replica in the set can handle requests, even if some nodes fail.

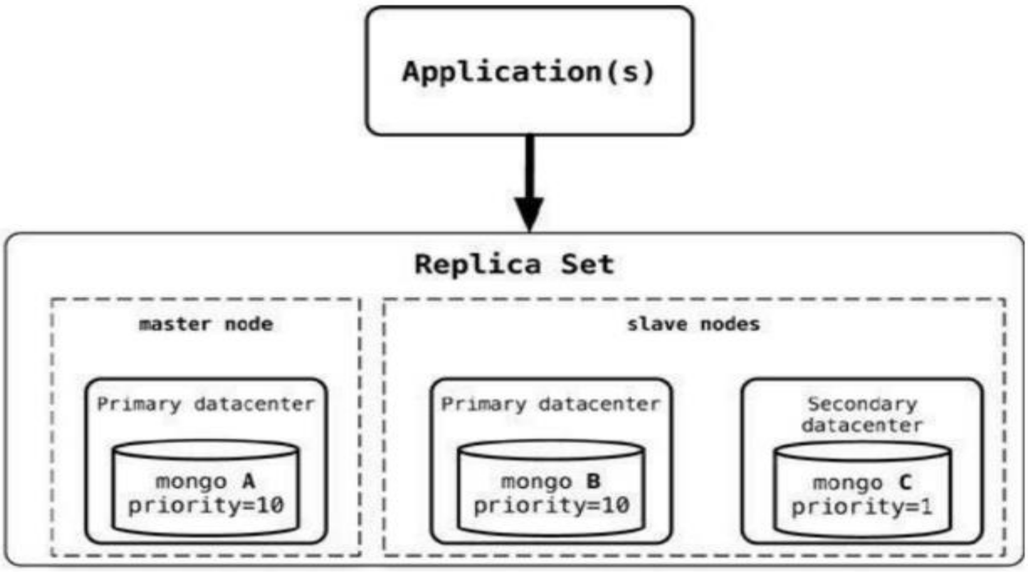
Key Points of Consistency and Availability in MongoDB:

1. **Consistency**: MongoDB provides consistency through **Write Concern** and **Read Concern** settings, ensuring that data is written to a majority of replica set members before being acknowledged. Reads are ensured to return the most recent data (with settings like majority).
2. **Availability**: MongoDB ensures availability through **Replication**. If the primary node goes down, one of the secondary nodes is automatically promoted to primary (election process), and the system continues to handle requests.

Replica Sets in MongoDB

In MongoDB, **Replica Sets** are used to ensure high availability and redundancy. A replica set consists of:

- **Primary Node**: The main node that handles all write operations.
- **Secondary Nodes**: Nodes that replicate the data from the primary and handle read operations, unless configured otherwise.
- **Arbiter (optional)**: A node that does not hold data but participates in elections to ensure a primary is always available in case of a failure.



Explanation of Replica Set Configuration:

- **Primary Node**: This is where all write operations occur. Writes are replicated to the secondary nodes.
- **Secondary Nodes**: These nodes replicate data from the primary node. By default, they handle read operations but can be configured to also become the primary if the current primary fails.
- **Arbiter**: The arbiter does not store data but is involved in elections to determine the new primary node if the current one fails. It ensures the system remains highly available by ensuring that the replica set has a majority for elections.

Consistency and Availability in Action

- **Write Concern**: When writing data, you can set the **write concern** to ensure that the data is acknowledged by a certain number of replica set members (e.g., w:majority). This ensures consistency by waiting for the write to propagate to a majority of nodes before confirming the write.
- **Read Concern**: MongoDB uses **read concern** to determine whether the data being read is the most recent version. For instance, with readConcern: "majority", MongoDB ensures that only data that has been acknowledged by the majority of the replica set is returned in the read query.
- **Replication and Failover**: MongoDB automatically handles failover. If the primary node becomes unavailable, one of the secondaries is promoted to primary, ensuring the system remains available.

2. With suitable diagrams, explain horizontal sharding in MongoDB for adding a new node to an existing replica-set and each shard is a replica set. [10] 4 L2

Scheme: Definition + Explanation + Example + Diagram – 2 + 3 + 2 + 3 Marks

Solution :

Horizontal Sharding in MongoDB

Horizontal sharding in MongoDB refers to the process of distributing data across multiple machines to spread the load and improve scalability. In this approach, the data is partitioned into smaller chunks and distributed across different shards, each of which can reside on a different server. Each shard contains a subset of the data and operates as a replica set to provide high availability and redundancy.

Steps for Horizontal Sharding in MongoDB:

Shard Key Selection:

The first step is to choose a shard key, which is used to determine how data will be distributed across the shards.

The shard key is typically a field that exists in all documents and determines how the data will be split into chunks.

Adding a New Node to an Existing Replica Set:

When you add a new shard (node) to an existing replica set in MongoDB, you are essentially adding another replica to manage the data load across a larger set of resources.

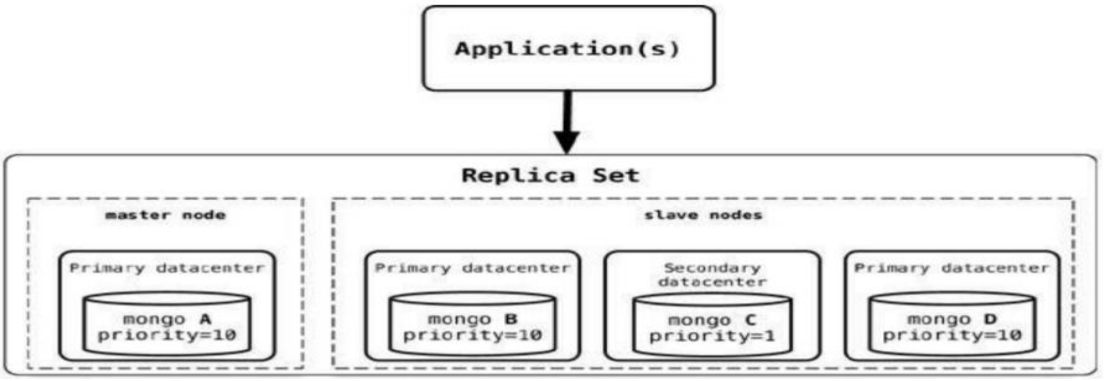
The new shard node is added to the sharded cluster, increasing the system's capacity to handle more data and queries.

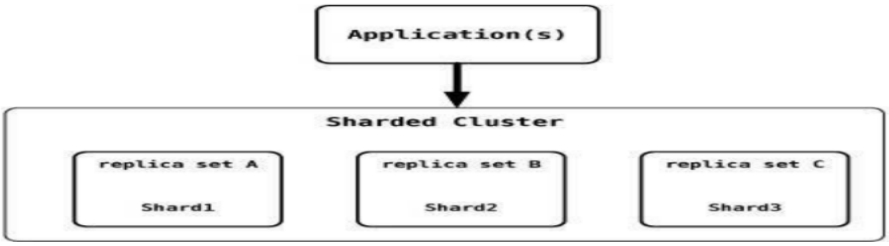
Components of Sharding in MongoDB:

- **Shard:** A shard is a single database partition, and each shard in a sharded MongoDB cluster is a replica set.
- **Config Servers:** These are used to store metadata about the sharded data (such as chunk distribution and shard keys).
- **Mongos (Query Routers):** The mongos process acts as a query router that routes

client requests to the correct shard based on the shard key.

- **Replica Set:** A replica set in MongoDB is a group of mongod instances that maintain the same data set. Each shard in the cluster is a replica set to ensure high availability.





	Config Servers: These servers store the metadata information for the sharded cluster. They track which shard stores which data (based on the shard key) and ensure that data is balanced across the shards. **Mongos Query Routers:** These are the front-end processes that receive client requests. They route the requests to the appropriate shard by looking up the shard key in the metadata stored in the config servers. **Shards (Replica Sets):** Each shard contains a subset of the data and is a replica set for redundancy and failover. Each shard can have one or more nodes (primary and secondary) to provide data availability. **Adding a New Node to the Replica Set (Shard):** **Step 1: Add a New Shard to the Cluster** The first step is to add the new shard to the cluster. This involves starting a new mongod process that will act as the new shard. This new shard is also configured as a replica set for high availability. **Command Example:** `sh.addShard("newShardReplicaSet/host:port")` **Step 2: MongoDB Balancer** Once the new shard is added, MongoDB's balancer will redistribute the data across all the shards based on the shard key. This ensures that the data is evenly distributed among the shards. **Balancer Role:** The balancer checks which shards are under-loaded or over-loaded and moves chunks accordingly. The balancer runs periodically to ensure that the data remains evenly distributed.			
3.	Describe the Query features and transactions of graph databases. **Scheme : Definition + Explanation + Example + Diagram – 2 + 3 + 2 + 3 Marks** **Solution :** **Query Features of Graph Databases** Graph databases are designed to efficiently store and process graph structures, such as nodes, edges, and properties. Their query capabilities focus on traversing relationships between entities. Below are key query features: 1. **Graph Traversal**: o **Traversal** is the most powerful feature in graph databases. It allows queries to follow relationships between nodes. For example, you can start with one node (e.g., a person) and traverse through the connected nodes (e.g., friends, coworkers) based on relationship types and properties. o Common queries include "find all friends of X" or "find the shortest path between two nodes." 2. **Pattern Matching**: o Graph databases use pattern matching to define relationships between nodes and edges. For example, you can query for a specific pattern like (A)-[:FRIEND]->(B), which represents a relationship between nodes A and B with the type FRIEND. o This is different from relational databases, where you need complex joins to traverse relationships. 3. **Depth and Path Queries**: o Depth refers to how many levels of relationships to traverse in a graph. Graph queries can be written to fetch relationships up to a specific depth or to find the shortest path between two nodes. o Path queries help find a direct or indirect connection between nodes based on relationships. 4. **Aggregations and Filtering**: o Graph databases allow aggregation and filtering on node properties or	[10]	3	L2

relationship attributes. For example, you can filter out nodes with specific properties (like age > 30) or calculate counts, averages, or other statistics across relationships.

5. **Flexible Schema**:
    - Unlike relational databases, graph databases do not require a fixed schema. This flexibility allows for dynamic querying of varied data structures without the need to redefine the schema when new types of relationships or nodes are added.

Transactions in Graph Databases

Graph databases also support **transactions**, which are crucial for ensuring data integrity and consistency in multi-user environments. Key aspects of transactions in graph databases are:

1. **ACID Properties**:
    - **Atomicity**: Transactions in graph databases are atomic, meaning they either complete fully or not at all. If an error occurs during the transaction, all changes are rolled back.
    - **Consistency**: The database is always in a valid state before and after a transaction. Constraints (such as unique relationships or properties) are maintained throughout the transaction.
    - **Isolation**: Transactions are isolated, meaning concurrent transactions do not interfere with each other, preventing issues like dirty reads or lost updates.
    - **Durability**: Once a transaction is committed, the changes are permanent and will survive system failures.
2. **Transaction Control**:
    - Graph databases provide mechanisms to manage transactions, such as **begin**, **commit**, and **rollback**. For example, in Cypher (a query language for Neo4j), you can start a transaction with BEGIN and commit with COMMIT.
    - Transactions can span multiple operations, ensuring that a series of related changes are either all applied or none at all.
3. **Concurrency Control**:
    - Graph databases implement concurrency control to handle multiple transactions happening simultaneously. This ensures that operations like graph traversal or relationship creation do not result in conflicts.
    - Optimistic or pessimistic concurrency control methods can be used depending on the graph database.
4. **Eventual Consistency**:
    - In distributed graph databases, some systems may support eventual consistency rather than strict ACID properties. This means that, while transactions are consistent across nodes, it may take some time for changes to propagate to all parts of the system.

4.	a. Elaborate the suitable and not suitable use cases of document databases.	[5]	4	L2

Scheme : Suitable case + unsuitable case+ example for each – 2+2+1 Marks

Solution :

Suitable Use Cases for Document Databases

1. **Content Management Systems (CMS):**
    - **Reason:** Document databases are ideal for storing diverse types of content (e.g., articles, blogs, or product descriptions) with varying structures. Each document can store data in a flexible, schema-less format, making it easy to update or change content without affecting the entire database.
    - **Example:** A CMS where articles have different metadata and text structures, and each article can have varying fields or tags.
2. **E-commerce Platforms:**
    - **Reason:** Products in an e-commerce platform often have diverse attributes (e.g., size, color, specifications). Document databases allow storing these variable attributes in a single document for each product.
    - **Example:** An online store where each product has different attributes, such as clothing sizes, colors, and technical specifications for electronics, which would be difficult to model in a traditional relational database.
3. **Real-Time Analytics and Monitoring:**
    - **Reason:** Document databases can handle large amounts of unstructured or semi-structured data that need to be processed quickly. They allow for the easy integration of various types of data from different sources in a flexible way.
    - **Example:** A real-time dashboard or logging system that needs to store varying data from different sensors or logs.
4. **Personalized User Profiles:**
    - **Reason:** Document databases can store user profiles with diverse and evolving attributes (e.g., preferences, behavior, or recommendations) that vary over time without altering the database schema.
    - **Example:** Storing user profiles for a recommendation system in a social media platform or e-commerce site.
5. **Mobile and Web Applications (with Dynamic Schemas):**
    - **Reason:** Mobile apps often require flexibility to evolve their data model quickly, such as changing features, adding or removing fields, or adjusting for different users.
    - **Example:** A mobile app where user preferences and settings are stored in a document format, making it easy to update without a rigid schema.

Not Suitable Use Cases for Document Databases

1. **Transactional Systems with ACID Requirements:**
    - **Reason:** Document databases may not provide full ACID compliance (Atomicity, Consistency, Isolation, Durability) in the way that relational databases do, which can be a drawback for applications requiring complex transactions.
    - **Example:** A banking system where multiple related operations need to be executed as part of a single transaction (e.g., transferring money between accounts).
2. **Highly Structured and Relational Data:**
    - **Reason:** Document databases are less suited for scenarios where data is highly structured and has many interrelationships. Relational databases are more effective for managing such data with complex joins and constraints.
    - **Example:** A university management system with complex relationships between students, courses, professors, and departments, where relational integrity is critical.
3. **Applications with Complex Queries Involving Joins:**
    - **Reason:** Document databases do not excel at performing complex joins or queries involving multiple collections of related data. While aggregation pipelines can help, they are not as efficient as relational joins.
    - **Example:** An enterprise application requiring frequent, complex reports that join multiple tables (e.g., sales orders, customers, products).
4. **Data Consistency Across Multiple Documents:**
    - **Reason:** In a document database, data consistency across documents (like foreign key relationships) can be hard to maintain, especially if there are complex interdependencies.

4. b. Explain some suitable use cases of graph databases and describe when we should not use graph databases [5]

Scheme- **Suitable case + unsuitable case+ example for each – 2+2+1 Marks**

Suitable Use Cases of Graph Databases:

1. **Social Networks:**
    - **Example:** Facebook, LinkedIn
    - **Why:** Social networks are based on interconnected data (users, posts, likes, comments, connections). Graph databases are perfect for modeling these complex relationships where the connections and patterns between users and their interactions are crucial.
2. **Recommendation Systems:**
    - **Example:** Amazon, Netflix
    - **Why:** Graph databases excel in recommendation engines because they can efficiently model relationships between products, users, ratings, and preferences. They can explore item similarities and user behaviors, recommending products or content based on connections in the graph.
3. **Fraud Detection:**
    - **Example:** Financial transactions, Banking Systems
    - **Why:** Graph databases can detect fraudulent patterns by modeling relationships between accounts, transactions, and users. For example, tracing a series of financial transactions can reveal hidden connections and anomalies that indicate fraudulent activity.
4. **Network and IT Operations:**
    - **Example:** Telecom Networks, IT Systems
    - **Why:** Graph databases help in modeling and analyzing network topologies and IT infrastructures. They allow efficient identification of network paths, device relationships, and potential vulnerabilities, which are difficult to represent using traditional relational databases.
5. **Knowledge Graphs:**
    - **Example:** Google's Knowledge Graph, Semantic Search
    - **Why:** Knowledge graphs help in storing and representing entities and the relationships between them. They are widely used in search engines for better understanding of user queries, context, and providing more relevant answers by connecting pieces of knowledge.
6. **Supply Chain Management:**
    - **Example:** Logistics, Inventory Systems
    - **Why:** In a supply chain, entities like suppliers, customers, inventory, and distribution points can be interconnected in complex ways. Graph databases allow for efficient tracking of inventory, shipments, and order relationships, identifying bottlenecks and optimizing routes.

When Not to Use Graph Databases:

1. **For Simple, Tabular Data:**
    - **Example:** Basic business records (e.g., employee details, customer information)
    - **Why Not:** Graph databases are ideal for complex relationships, but if your data is simple, relational, and doesn't involve intricate connections, relational databases or NoSQL solutions like document-based databases (e.g., MongoDB) are more suitable due to their simpler structure and ease of querying.
2. **When Data Integrity and ACID Transactions Are Critical:**
    - **Example:** Banking Systems, Financial Accounting
    - **Why Not:** While some graph databases support ACID transactions, they are not as optimized as relational databases, which are designed for scenarios requiring strong data consistency and integrity across complex transactions. Relational databases are better suited for these use cases.
3. **When You Need High-Volume, High-Speed Writes/Reads with Minimal Relationships:**
    - **Example:** Logging, Real-time Analytics
    - **Why Not:** If the application involves handling vast amounts of simple records (e.g., logging or time-series data), traditional databases or key-value stores (e.g., Redis, Cassandra) may offer better performance than graph databases, which are optimized for complex, interconnected data.
4. **For Complex Analytical Queries:**
    - **Example:** Large-scale Data Warehousing

- **Why Not:** While graph databases excel in relationship-based queries, for complex analytical queries involving aggregation, multi-dimensional analysis, and complex joins (e.g., large-scale reporting), relational databases or columnar databases (e.g., Amazon Redshift) are often more efficient.

5. **When There Is No Need for Relationships Between Data Points:**
    - **Example:** Storing configuration data, simple key-value pairs
    - **Why Not:** Graph databases are designed for relationship-rich data. If your use case involves storing independent, non-relational data, NoSQL document databases or traditional relational databases would be simpler and more efficient.

5.	With the context of graph database, explain the following: i. Mechanism of relationships with properties ii. Application level sharding	[10]	3	L2

Scheme : Defination+explanation+example= (2+2+1)+(2+2+1)

i. Mechanism of Relationships with Properties in Graph Databases

In a **graph database**, relationships (or edges) connect nodes (or vertices) and represent the associations or connections between them. Relationships are not just pointers from one node to another; they can also have properties (key-value pairs) that provide additional context about the relationship.

- **Structure:** A relationship is always directed (having a start and an end node) and can be of various types, such as FRIEND, COLLEAGUE, LIKES, etc. These types define the kind of connection between nodes.
- **Properties on Relationships:** Properties can be assigned to relationships to describe specific details about the connection. For example, a relationship between two people (nodes) could have properties like since (a date), strength (numeric value), or location (string), representing attributes specific to the connection.

Example:

cypher

Copy code

```
CREATE (a:Person {name: 'Alice'})-[:FRIEND {since: 2020, strength: 8}]->(b:Person {name: 'Bob'})
```

In this case:

- Alice and Bob are nodes (people).
- The relationship FRIEND connects them.
- The relationship has properties: since: 2020 and strength: 8.

- **Querying Relationships with Properties:** Relationships can be queried to filter based on their properties. For example, you could find all friends with a specific strength or all relationships created after a certain year:

```
MATCH (a:Person)-[r:FRIEND]->(b:Person)
WHERE r.since > 2019
RETURN a, b, r
```

This mechanism allows graph databases to efficiently model and store rich, dynamic relationships with context, making them ideal for applications like social networks, recommendation engines, and knowledge graphs.

ii. Application Level Sharding in Graph Databases

Sharding is the process of distributing data across multiple servers or machines to ensure scalability and performance, especially when dealing with large datasets. **Application-level sharding** in graph databases refers to the practice of manually managing how the data is split or partitioned across different database instances at the application layer, rather than relying solely on the database itself to handle the distribution.

- **Why Application-Level Sharding?** Graph databases store highly interconnected data, and relationships between nodes often span multiple partitions. This makes sharding a challenging task, as a query that involves multiple nodes and their relationships may span across different shards. With application-level sharding, the application itself manages how data is distributed, giving more control over how to shard based on the specific use case.
- **How It Works:**
    1. **Shard Key Selection:** The application selects a **shard key** (e.g., a user ID, geographic location, or a business entity) to determine how nodes and

relationships are distributed across shards.

2. **Data Distribution:** Nodes and relationships are then placed on different servers based on the shard key. For example, all nodes related to a particular user or geographical region could reside on the same shard.
3. **Cross-Shard Queries:** When a query involves nodes or relationships on different shards, the application is responsible for ensuring that queries span the relevant shards and aggregate the results.

- **Advantages of Application-Level Sharding:**
    - **Fine-grained control** over data distribution.
    - **Customization** for specific access patterns, such as when certain users or entities are frequently queried together.
    - Can **scale horizontally** by adding more servers or nodes as data grows.
- **Challenges:**
    - **Increased complexity**: The application must be responsible for managing shard distribution, ensuring efficient query routing, and handling cross-shard joins.
    - **Data consistency**: Maintaining consistency across shards, especially when relationships span multiple shards, can be challenging.
    - **Performance overhead**: If queries span multiple shards, the application may face performance issues due to the need to query and merge data across shards.

6.	a. Generate equivalent MongoDB queries for given SQL queries: i) SELECT * FROM order ii) SELECT * FROM order WHERE customerId=”BL_12182”. iii) SELECT orderId,orderDate FROM order WHERE customerID is BL_12182. iv) SELECT * FROM customerOrder, orderItem, product WHERE customerOrder.orderID=orderItem.customerOrderID AND orderItem.productID-product.productID AND product.name LIKE ‘%Refactoring%	[6]	4	L3

Scheme : Query writing and Explanation– 3+3 Marks

i)SQLQuery:

SELECT * FROM order;

MongoDB Equivalent:

db.order.find({});

ii)SQLQuery:

SELECT * FROM order WHERE customerId = "BL_12182";

MongoDB Equivalent:

db.order.find({ customerId: "BL_12182" });

iii)SQLQuery:

SELECT orderId, orderDate FROM order WHERE customerID = "BL_12182";

MongoDB Equivalent:

db.order.find({ customerId: "BL_12182" }, { orderId: 1, orderDate: 1, _id: 0 });

iv)SQLQuery:

SELECT * FROM customerOrder, orderItem, product WHERE customerOrder.orderID = orderItem.customerOrderID AND orderItem.productID = product.productID AND product.name LIKE '%Refactoring%'

MongoDB Equivalent:

```
db.customerOrder.aggregate([
  {
    $lookup: {
      from: "orderItem",
      localField: "orderID",
      foreignField: "customerOrderID",
      as: "orderItems"
```

```
    }
  },
  {
    $unwind: "$orderItems"
  },
  {
    $lookup: {
      from: "product",
      localField: "orderItems.productID",
      foreignField: "productID",
      as: "products"
    }
  },
  {
    $unwind: "$products"
  },
  {
    $match: {
      "products.name": { $regex: "Refactoring", $options: "i" }
    }
  }
]);
```

b. Compute cipher queries to:
i) find all outgoing relationships with the type of FRIEND, and return the friends' names of "AAAAAA" for depth=1
ii) Find relationships where a particular relationship property exists. Filter on the properties of relationships and query if a property exists or not.

[4]

Scheme : Explanation– 2+2 Marks

i) Find all outgoing relationships with the type of FRIEND, and return the friends' names of "AAAAAA" for depth=1:

```
MATCH (a:Person {name: "AAAAAA"})-[:FRIEND]->(b:Person)
RETURN b.name
```

- **Explanation:** This query matches all outgoing FRIEND relationships from the node where the name property is "AAAAAA". It then returns the name of the connected Person nodes (the friends) for depth=1.

ii) Find relationships where a particular relationship property exists. Filter on the properties of relationships and query if a property exists or not:

```
MATCH (a:Person)-[r:FRIEND]->(b:Person)
WHERE EXISTS(r.property_name)
RETURN r
```

- **Explanation:** This query matches all FRIEND relationships between Person nodes and filters the relationships where a particular property (e.g., property_name) exists. The EXISTS() function checks whether the specified property exists in the relationship. The query returns the relationships where the property is present.

Faculty Signature CCI Signature HOD Signature