GROUP MEMBER:Yue Ma,Yuheng Li and Conger Yang

1.

a.

Atomicity: In Neo4j, a transaction is an indivisible unit of work that either fully executes or not at all. This means that if any operation within a transaction fails, the entire transaction will roll back as if it had never started.

Consistency: Neo4j ensures that the execution of a transaction will transition the database from one consistent state to another. No rules or constraints of the database are violated during the execution of a transaction.

Isolation: Transactions in Neo4j are isolated, meaning that concurrently executing transactions do not interfere with each other. Each transaction is executed as if it were in an independent environment, invisible to other transactions until the transaction is completed.

Durability: Once a transaction is committed, its modifications to the database are permanent. Even in the event of a system failure, the results of the transaction will not be lost.

b.

Relational Databases: In relational databases, data relationships are maintained through foreign key constraints, and these relationships are not first-class citizens of the database. Querying relationships often requires multi-table join operations, which can become inefficient in complex relationship queries.

MongoDB: MongoDB is a document-oriented database that allows for nested documents and arrays, making it more flexible when dealing with data that has nested relationships. However, it still requires querying the nested structures within documents to understand and utilize relationships, which may not be as intuitive as a graph database.

Neo4j: Neo4j treats relationships as first-class citizens, where data relationships are as important as the data itself. In Neo4j, relationships are directly stored in the database, meaning that related nodes are physically linked, and accessing these relationships is as immediate and fast as accessing the data itself.

c.

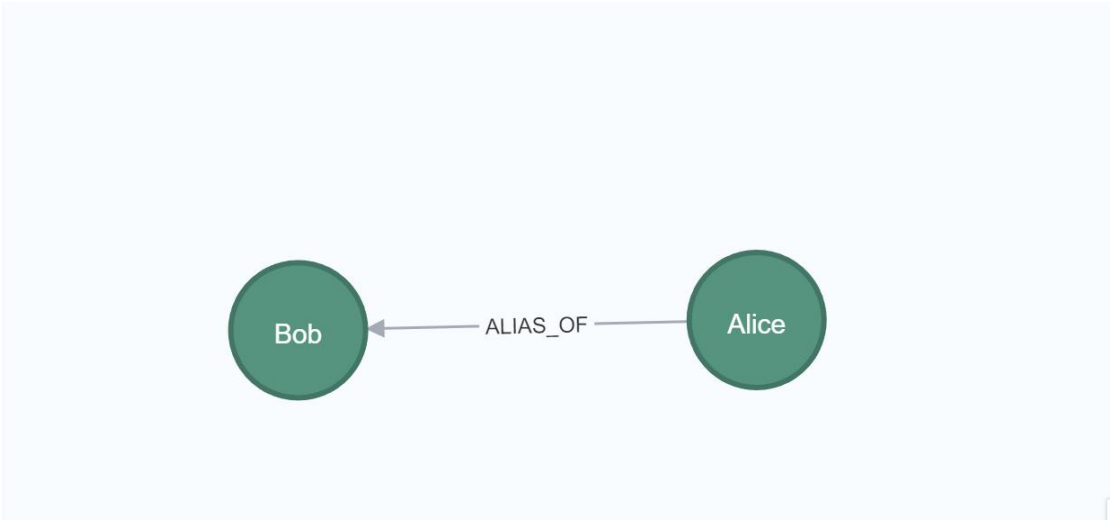
Non-Native Storage: Non-native storage refers to the storage of graph data in databases that are not specifically designed for graph data, such as relational databases, columnar storage, or other NoSQL databases. These systems may require additional indexing and query optimization to handle graph data, which can lead to inefficient querying.

Native Storage: Native graph storage is specifically built for storing graph data. In native graph storage, such as in Neo4j, data and their relationships are stored together, and because related nodes are physically linked in the database, accessing these relationships is as immediate and fast as accessing the data itself. This storage method typically offers faster query speeds and better performance, especially when dealing with complex graph traversals and large-scale graph data.

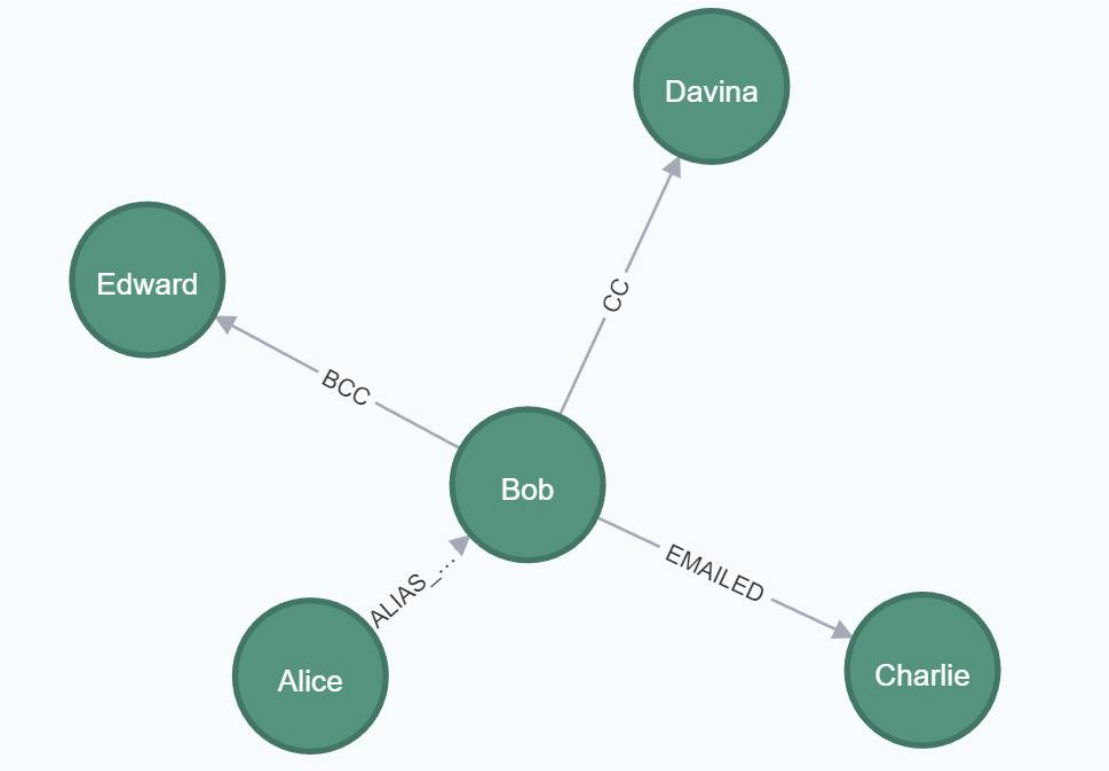
Impact on processing performance: Native graph storage, due to its optimized storage structure and indexing mechanism, usually provides faster query speeds and better performance, especially when handling complex graph traversals and large-scale graph data. Non-native storage may require more computing resources and time to process the same queries because they need to retrieve and connect data in non-optimized storage structures.

2.

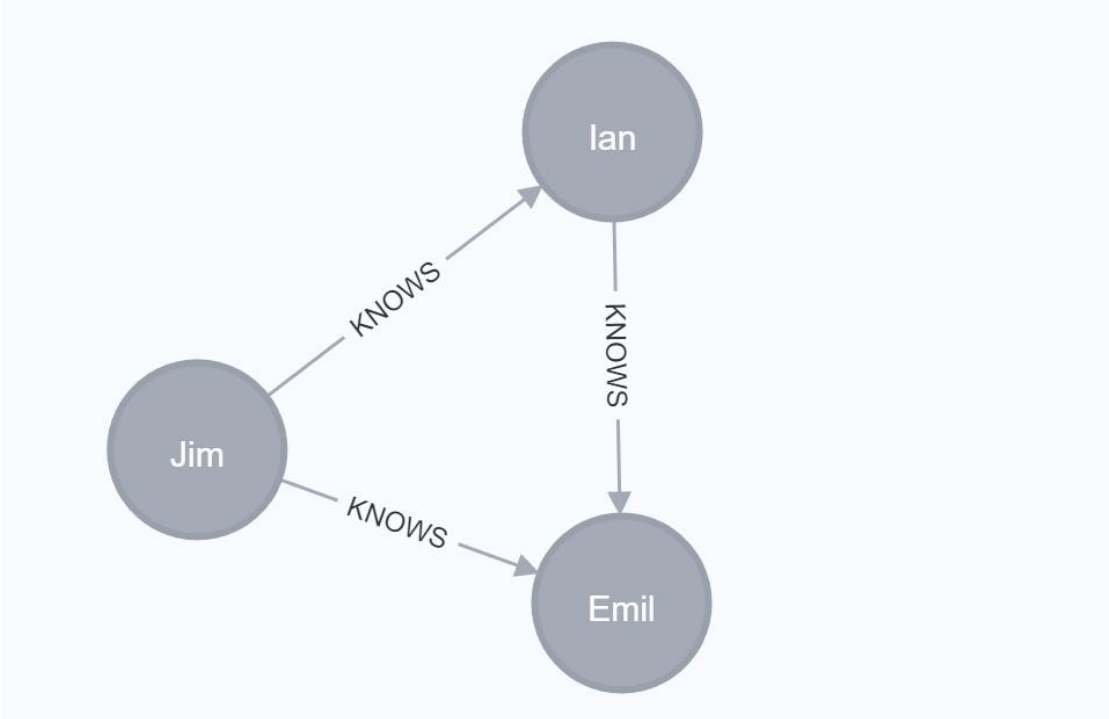
a.



b.



3.



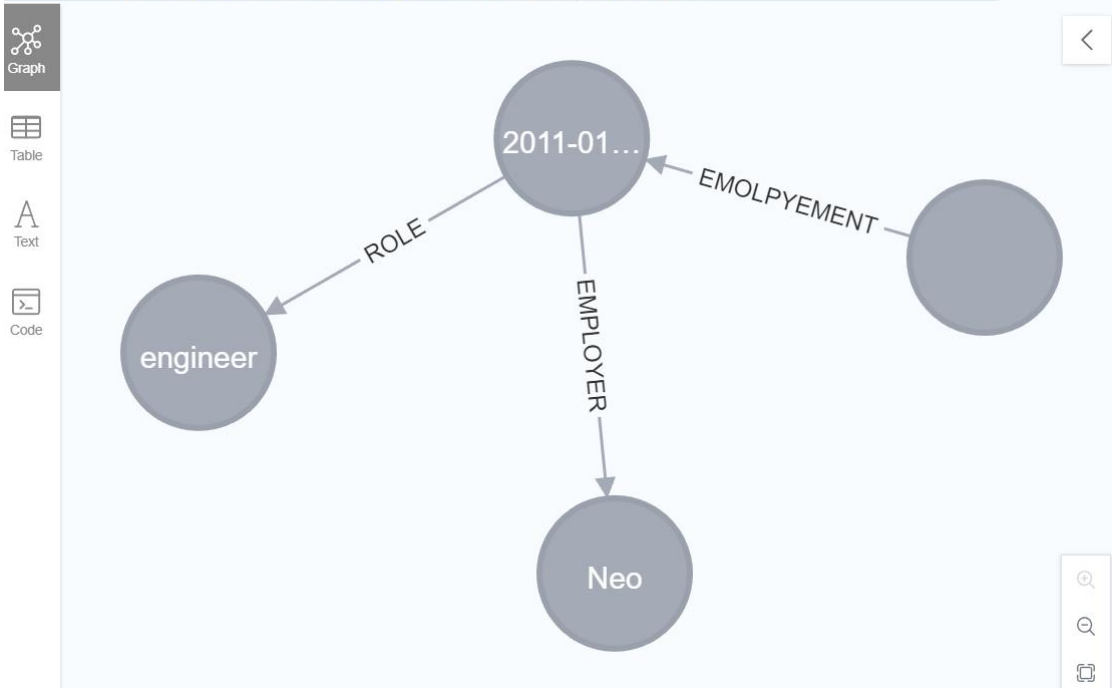
Explain the effect of the following script:

```
MATCH(a:Person {name: 'Jim' })-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
RETURN b, c
```

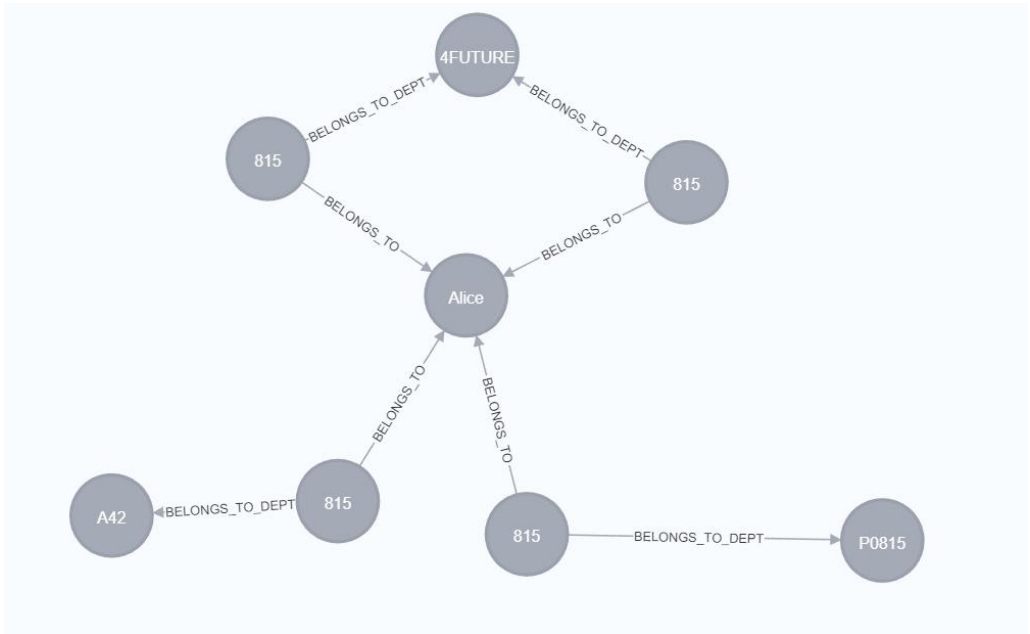
Establish the relationship between 'Jim' and b and use the KNOWS link, on this basis, establish the relationship between b and c and use the KNOWS link, and finally establish the relationship between a and c and use the KNOWS link to form a close loop of the relationship. Return b and c

4.

```
neo4j$ create(n:Company{name:'Neo'}),
       (l:Person{NAME:'lan'}),
       (e:Role{name:'engineer'}),
       (t:Job{start:'2011-01-05'}) create(t)-
       [tn:EMPLOYER]→(n),(l)-[lt:EMOLPYEMENT]→(t),(t)-
       [te:ROLE]→(e) return n,l,t,e,tn,lt,te
```



5.



```
MATCH (p:Persons {name: "Alice"}), (d:Department {name: "P0815"})
CREATE (m:Dept_Members {member: 815})
CREATE (m)-[:BELONGS_TO]->(p)
CREATE (m)-[:BELONGS_TO_DEPT]->(d)

MATCH (p:Persons {name: "Alice"}), (d:Department {name: "4FUTURE"})
CREATE (m:Dept_Members {member: 815})
CREATE (m)-[:BELONGS_TO]->(p)
CREATE (m)-[:BELONGS_TO_DEPT]->(d)

MATCH (p:Persons {name: "Alice"}), (d:Department {name: "A42"})
CREATE (m:Dept_Members {member: 815})
CREATE (m)-[:BELONGS_TO]->(p)
CREATE (m)-[:BELONGS_TO_DEPT]->(d)
```

Comparison

In Neo4j, traversing relationships is more efficient as they are directly stored as edges rather than needing join operations across tables.

Relational databases are efficient for structured, transactional operations, while Neo4j is optimized for querying interconnected data.

6.

(a)

This script retrieves the `FRIEND_OF` relationship between Sally and John and returns the value of the `since` property, indicating when they became friends.

(b)

i)

```
MATCH (book:Book {title: 'Moby Dick'})<-[:RATED]-(review)
RETURN avg(review.rating) AS average_rating
```

ii)

```
MATCH (author:Person)-[:WROTE]->(book:Book {title: 'Moby Dick'})
RETURN author.name AS authors
```

iii)

```
MATCH (sally:Person {name: 'Sally'})
RETURN sally.age AS age
```

iv)

```
MATCH (sally:Person {name: 'Sally'})-[:READ]->(book:Book {title: 'Moby Dick'})
MATCH (john:Person {name: 'John'})-[:READ]->(book)
```

```
RETURN CASE
    WHEN sally.read_date < john.read_date THEN 'Sally'
    ELSE 'John'
END AS first_reader
```