

# Pouring Coffee into the Matrix

## Java applications on Neo4j

Jennifer Reif

Email: [jennifer.reif@neo4j.com](mailto:jennifer.reif@neo4j.com)

Twitter: @JMReif

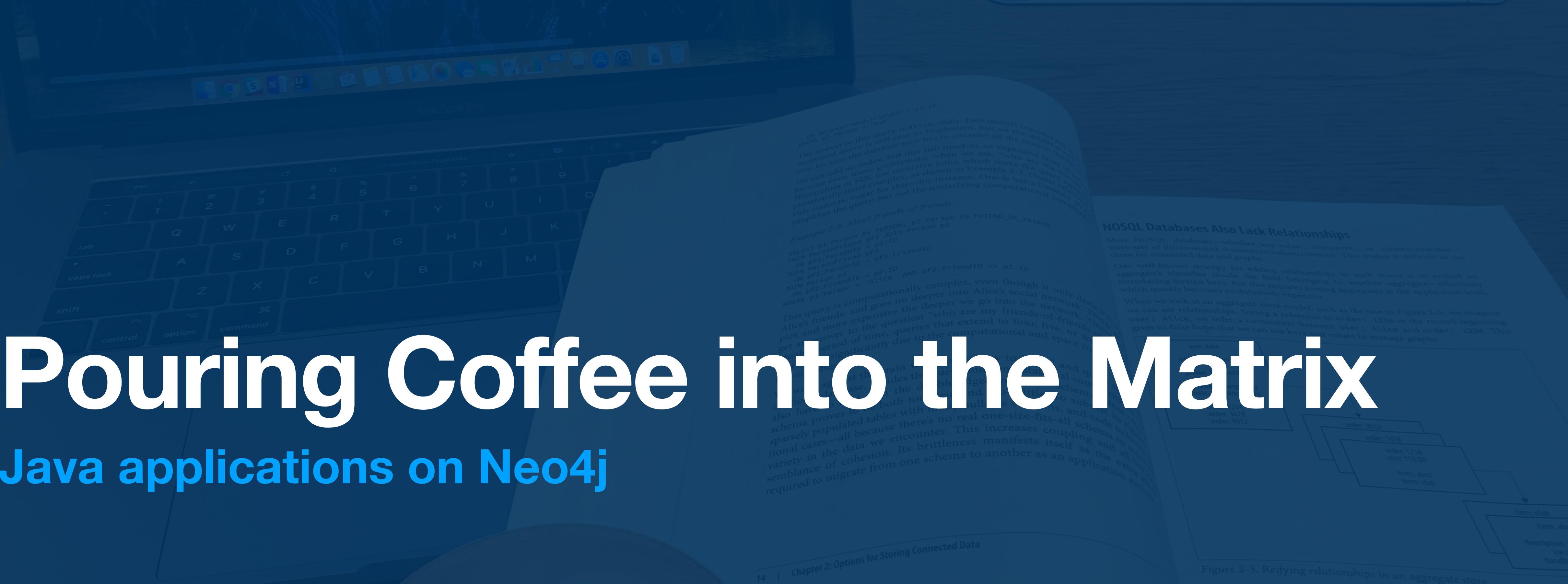
LinkedIn: [linkedin.com/in/jmhreif](https://linkedin.com/in/jmhreif)

Github: [github.com/JMReif](https://github.com/JMReif)

Website: [jmhreif.com](http://jmhreif.com)

Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really re-aggregates elsewhere in the database. But turning these inferred structure doesn't come for free, because relationships between class citizens in the data model—most aggregate stores fur-



# Who Am I?

- Developer + Advocate
- Continuous learner
- Technical content writer
- Conference speaker
- Other: geek

**Jennifer Reif**

Email: [jennifer.reif@neo4j.com](mailto:jennifer.reif@neo4j.com)

Twitter: [@JMHRrif](https://twitter.com/JMHRrif)

LinkedIn: [linkedin.com/in/jmhreif](https://linkedin.com/in/jmhreif)

Github: [github.com/JMHRrif](https://github.com/JMHRrif)

Website: [jmhreif.com](http://jmhreif.com)



...  
+ Add another card  
MacBook Pro  
Search or type URL  
tab  
caps lock  
shift  
command  
14 | Chapter 2: Options for Storing Connected Data  
The answer to this query is Alice; sadly, Zach doesn't have any friends. This is because the database now has recursive relationships, so we can add an index, but this still becomes even more problematic with hierarchies in SQL, use recursive hierarchical syntax for this—for instance, this query is much simpler:  
*Example 2-3. Alice's friends-of-friends*  
This query is computationally complex and more expensive than the previous one, as it needs to get an answer to the question "who are Alice's friends of friends?" over a reasonable period of time, queries that deteriorate significantly due to the joining tables.  
We work against the grain when working with a relational database. Besides the query, we also have to deal with the schema, which proves to be both too rigid and too sparse in populated tables with many relational cases—all because there's no semblance of cohesion. Its brittleness required to migrate from one schema to another.

# What is graph?

14 | Chapter 2: Options for Storing Connected Data

The answer to this query is Alice; sadly, Zach doesn't consider himself reciprocal. The query is still easy to implement, but on the database now has to consider all the rows in the PersonFriend table. We can add an index, but this still involves an expensive layer of memory. Hierarchies in SQL use recursive joins, which makes the query even more complex. Oracle has a CONNECT BY clause that provides syntactic sugar for this—for instance, Oracle has a COMPUTATIONAL complexity of 2<sup>n</sup> for this query, but not for the underlying computational complexity of the query.

```
Example 2-3. Alice's friends-of-friends
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID
JOIN Person p2
ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID => p1.ID
```

This query is computationally complex, even though it only deals with

Alice's friends, and goes no deeper into Alice's social network. Things

get an answer to the question "who are my friends-of-friends," but over a

reasonable period of time, queries that extend to four, five, or six degrees

deteriorate significantly due to the computational and space complexity

of joining tables.

We work against the grain whenever we try to model and query connected

data in a relational database. Besides the query and computational complexity, we also have to deal with the double-edged sword of schema. More rigid schema proves to be both too rigid and too brittle. To subvert its rigidity, we turn to sparsely populated tables with many nullable columns, and code to handle computational cases—all because there's no real one-size-fits-all schema to handle the variety in the data we encounter. This increases coupling and all but eliminates semblance of cohesion. Its brittleness manifests itself as the extra complexity required to migrate from one schema to another as an application evolves.

## NOSQL Databases Also Lack Relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

When we look at an aggregate store model, such as the one in Figure 2-3, we imagine we can see relationships. Seeing a reference to order: 1234 in the record beginning user: Alice, we infer a connection between user: Alice and order: 1234. This gives us false hope that we can use keys and values to manage graphs.

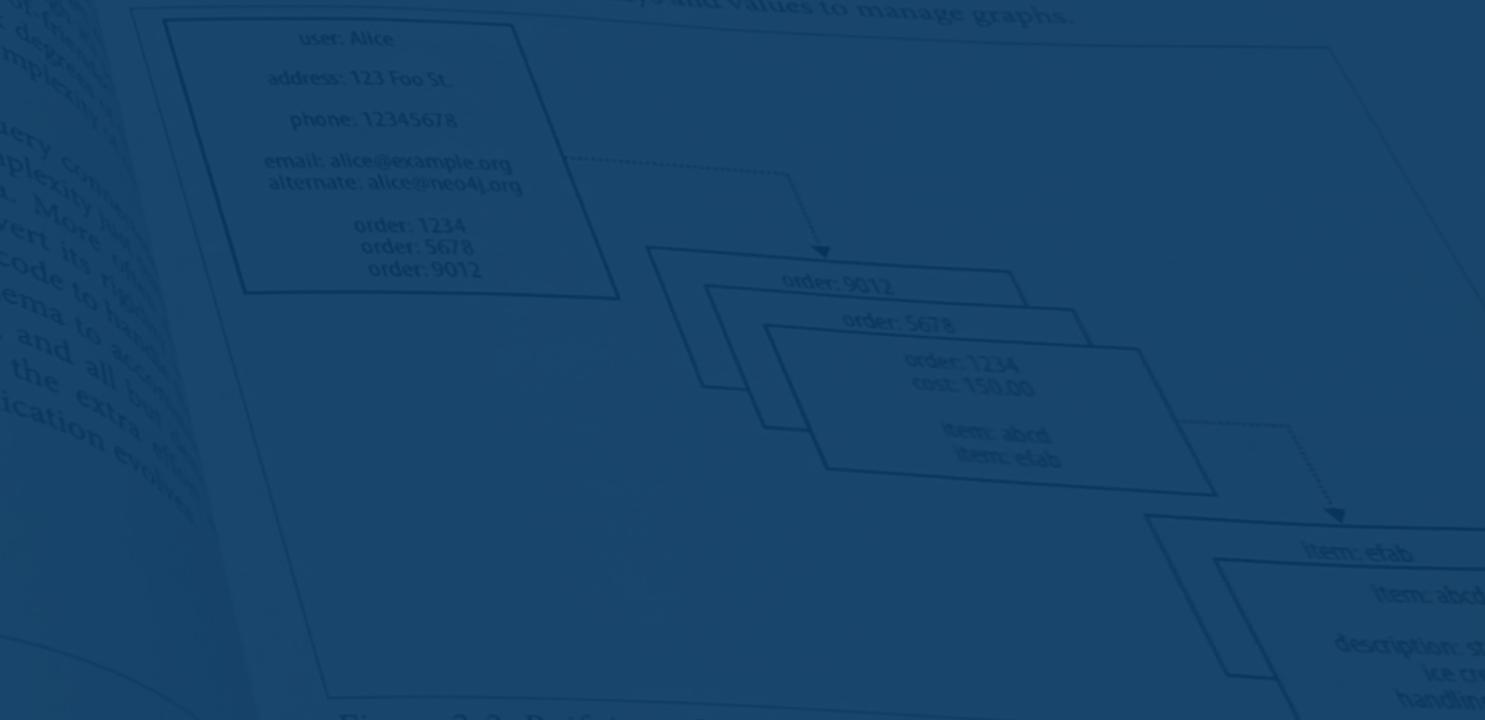
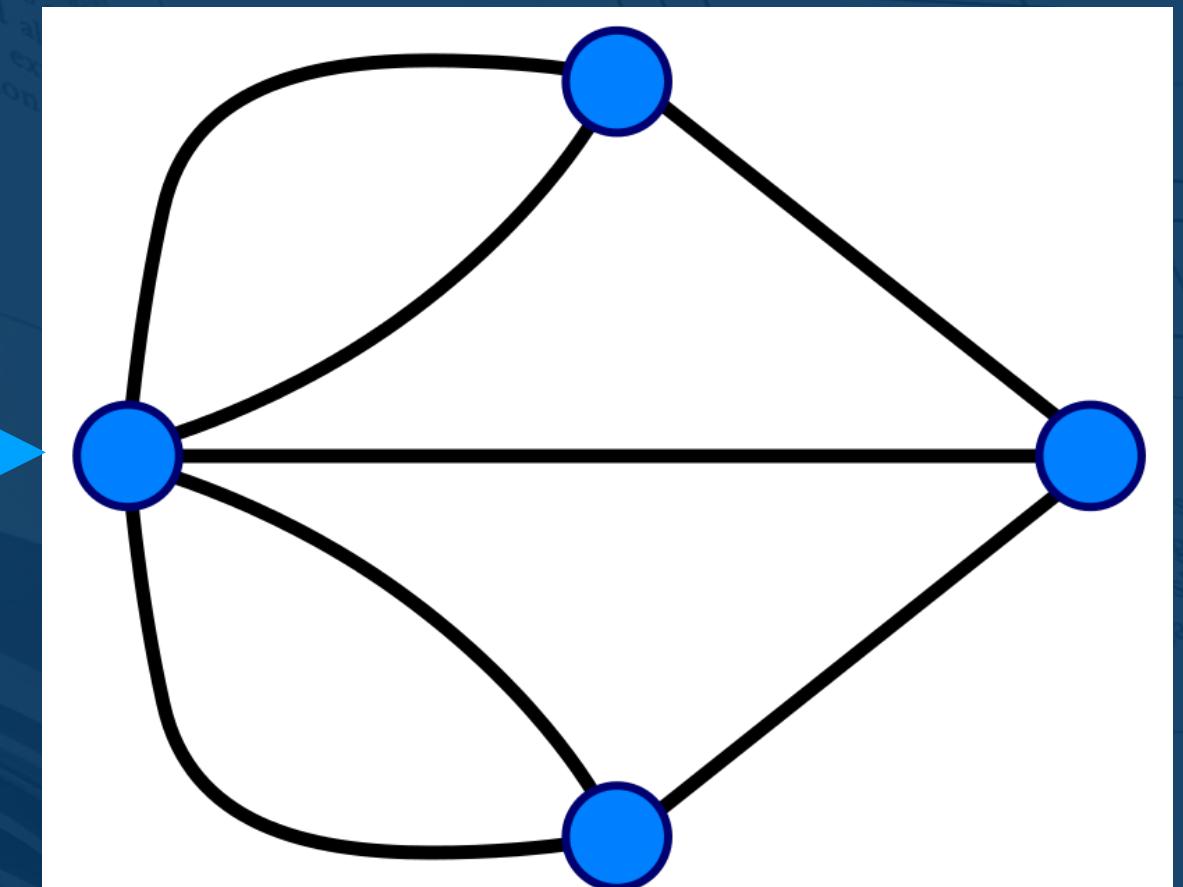
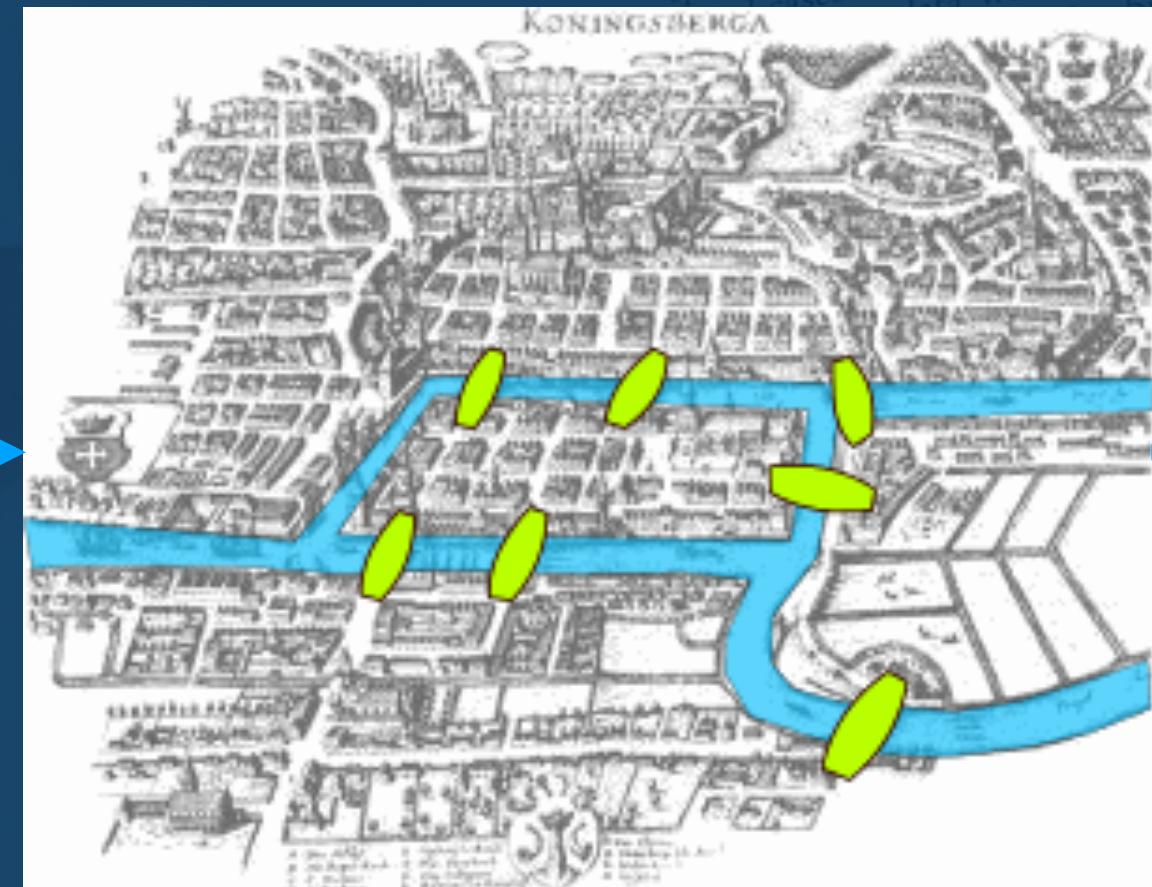
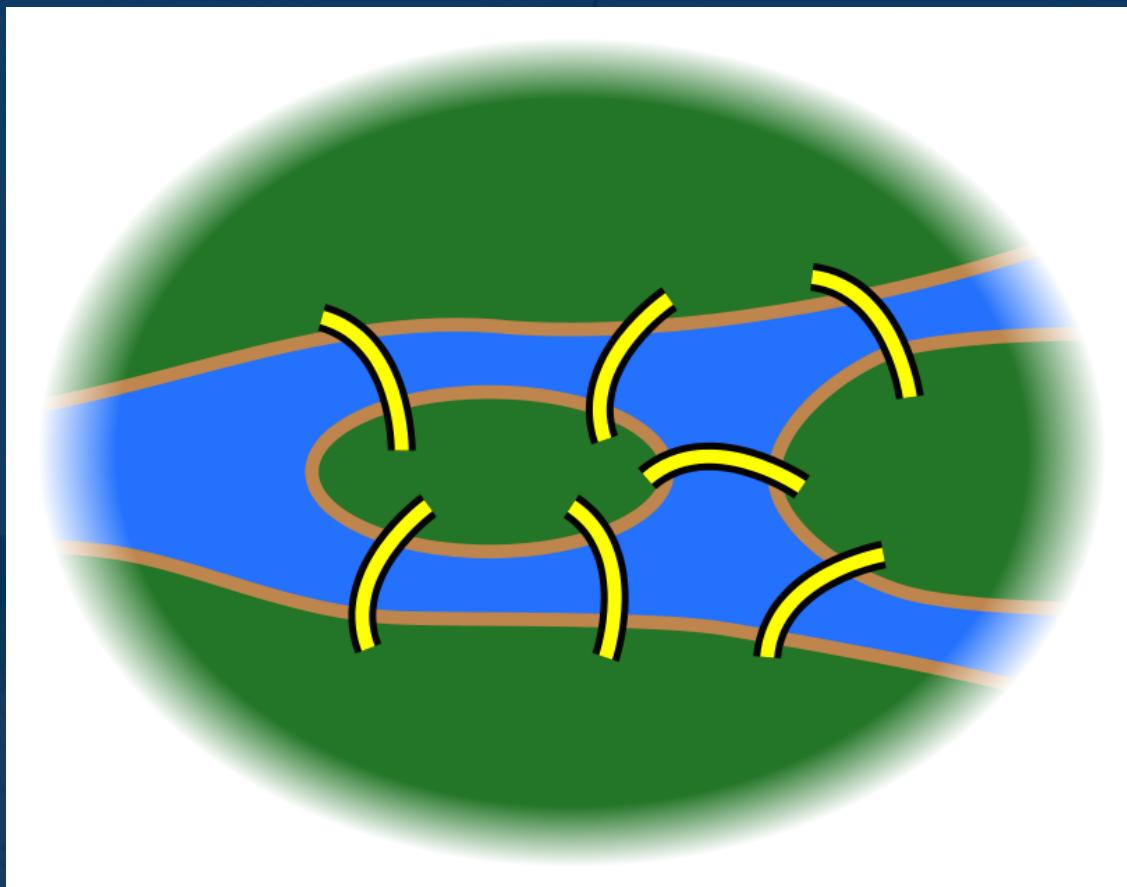


Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really references to aggregates elsewhere in the database. But turning these inferred structure doesn't come for free, because relationships between class citizens in the data model—most aggregate stores fur-

# What is a graph?

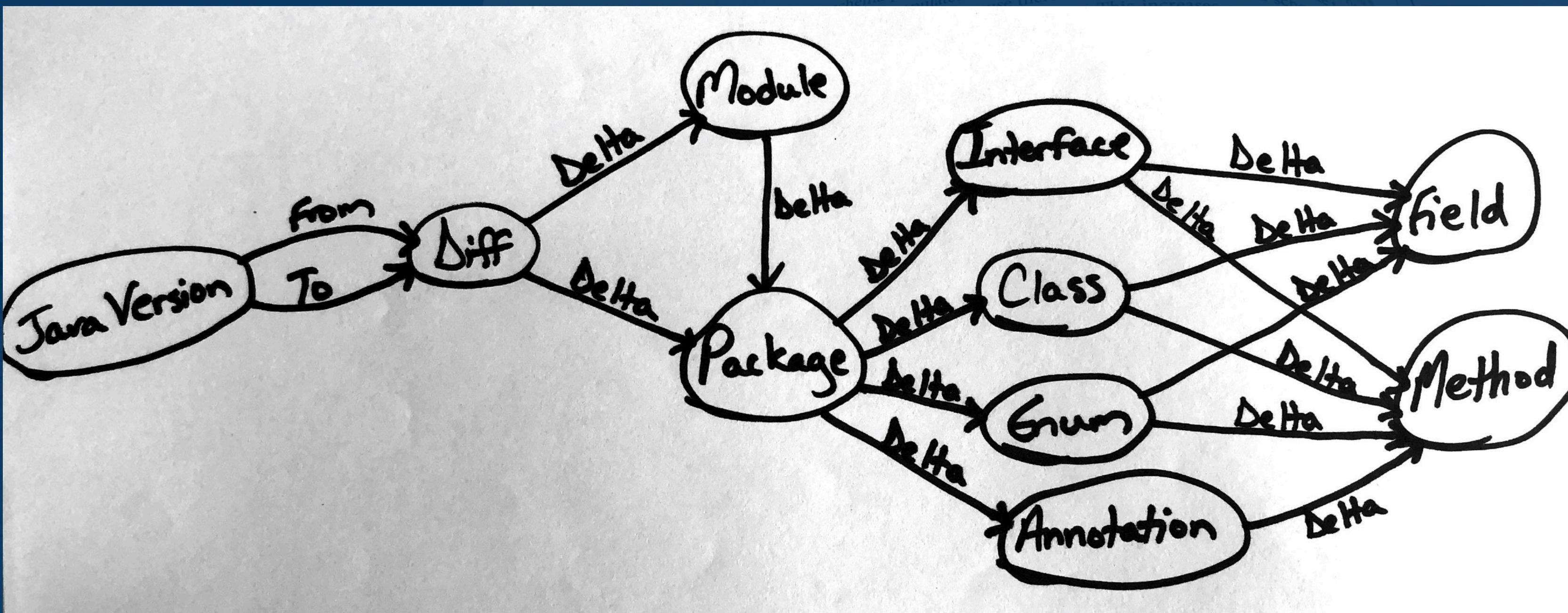
- Graph much older than technology of the last decade...
- Arose from solution need (then and now)



Seven Bridges of Königsberg problem. Leonhard Euler, 1735

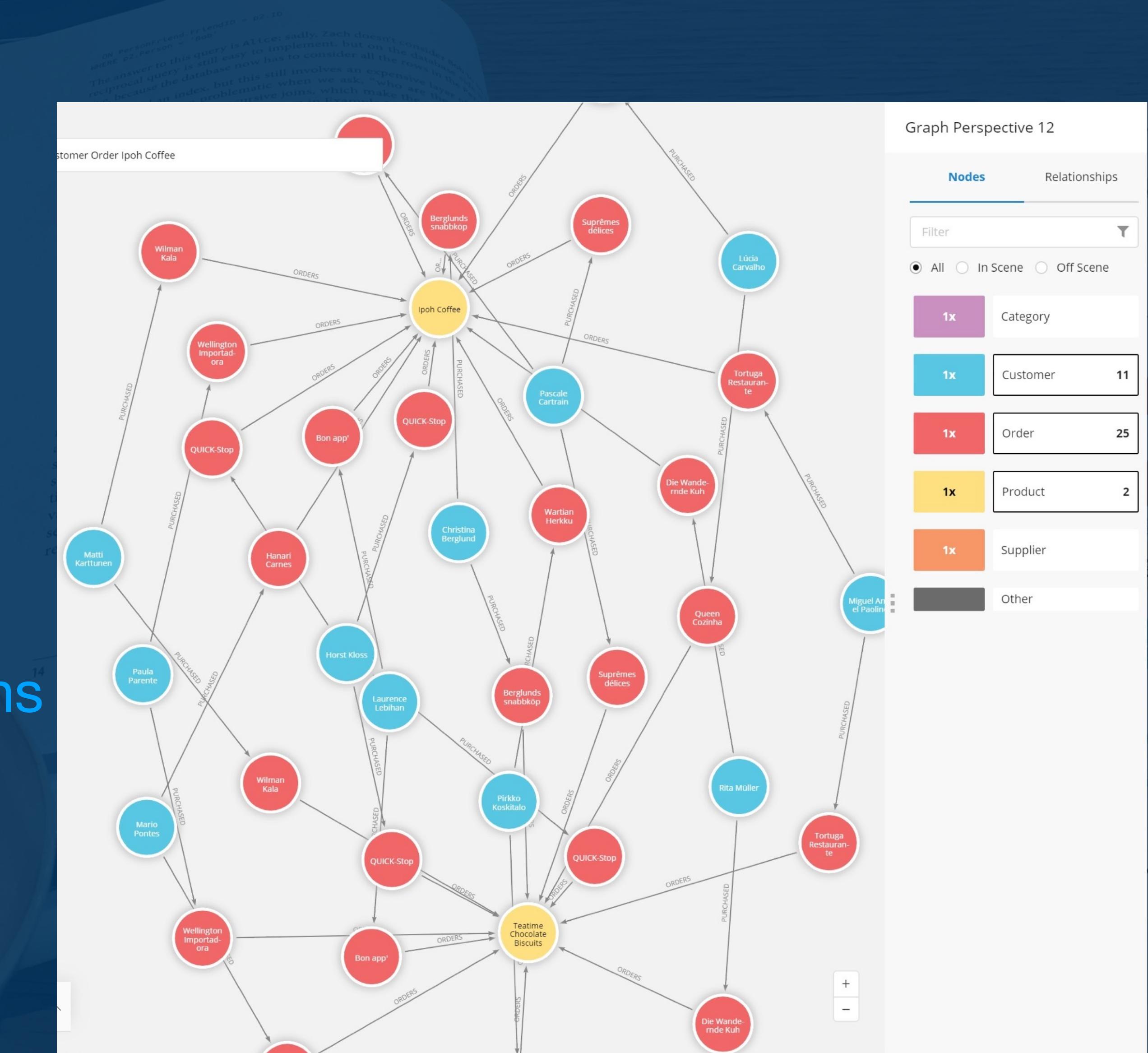
# Java domain

- Find field changes between Java 17 and Java 8?
- Update doc URL for a Class
- Find most similar diffs (which affected same components)



# Enter stage right - graphs

- Document path (not just data)
- Answering how and why
- Understanding connections
- Find alternates, impacts, etc
- Produce better assumptions, decisions



# Understanding the model



14 | Chapter 2: Options for Storing Connected Data

The answer to this query is Alice; sadly, Zach doesn't consider himself reciprocal. The query is still easy to implement, but on the database now has to consider all the rows in the PersonFriend table. We can add an index, but this still involves an expensive query. Some databases provide even more problematic when we ask, "Who are my friends-of-friends?" Hierarchies in SQL use recursive joins, which makes the query exponentially more complex, as shown in Example 2-3. Some relational databases provide syntactic sugar for this—for instance, Oracle has a CONNECT BY clause that simplifies the query, but not the underlying computational complexity.

**Example 2-3. Alice's friends-of-friends**

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID
JOIN Person p2
ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID => p1.ID
```

This query is computationally complex, even though it only deals with Alice's friends, and goes no deeper into Alice's social network. Things get an answer to the question "who are my friends-of-friends?" in a reasonable period of time, queries that extend to four, five, or six degrees deteriorate significantly due to the computational and space complexities of joining tables.

Work against the grain whenever we try to model and query a connected database. Besides the query and computational complexity, we have to deal with the double-edged sword of schema evolution. Schema evolution is hard to do in a disconnected database, but it's even harder in a connected one. If we want to update a table with new columns, we have to update all the other tables that reference it. This causes a loss of data variety in the database, which is the opposite of what we want. This semblance of cohesion is its primary downside, as the extra effort required to migrate from one schema to another as an application evolves.

## NOSQL Databases Also Lack Relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

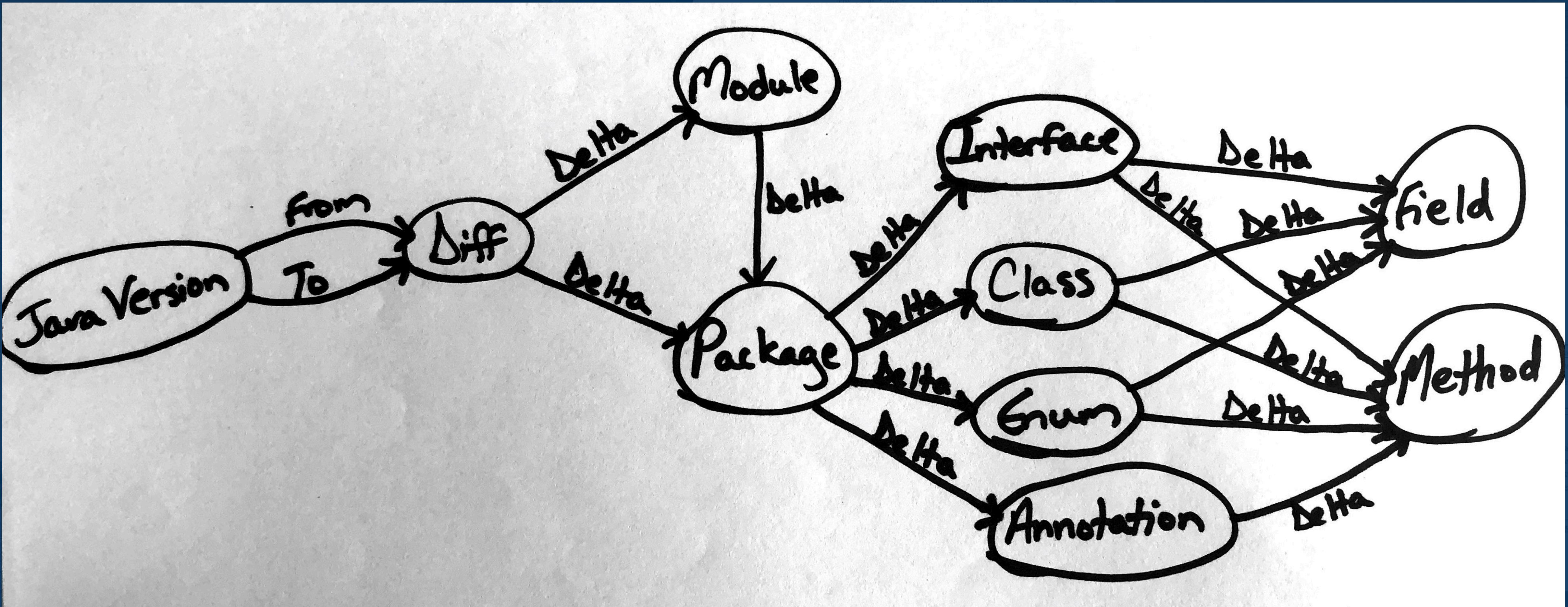
When we look at an aggregate store model, such as the one in Figure 2-3, we imagine we can see relationships. Seeing a reference to `order: 1234` in the record beginning `user: Alice`, we infer a connection between `user: Alice` and `order: 1234`. This gives us false hope that we can use keys and values to manage graphs.

```
user: Alice
address: 123 Foo St.
phone: 12345678
email: alice@example.org
alternate: alice@neo4j.org
order: 1234
order: 5678
order: 9012

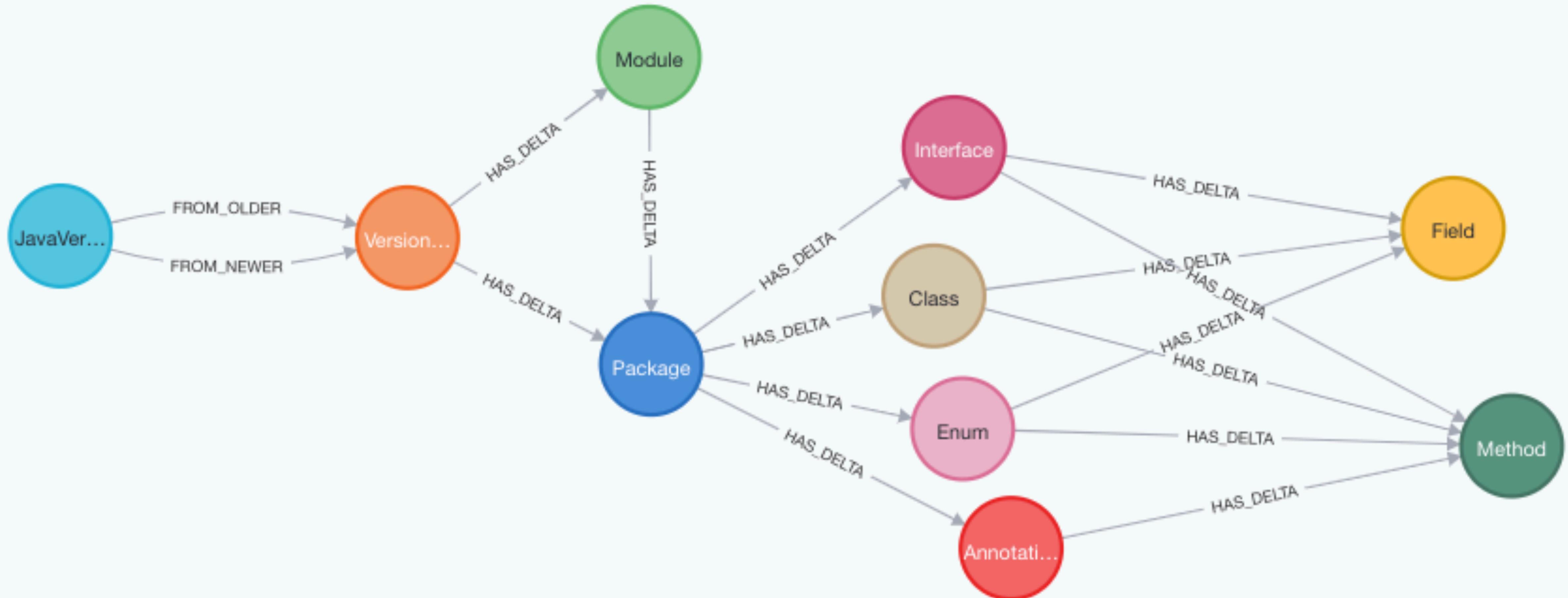
order: 9012
order: 5678
order: 1234
user: Alice
item: abcde
item: abcde

item: abcde
item: abcde
description: ...
handling: ...
```

# Whiteboard friendliness



# Graph data



# Graph components

- Node (vertex)
    - Objects

# Graphs

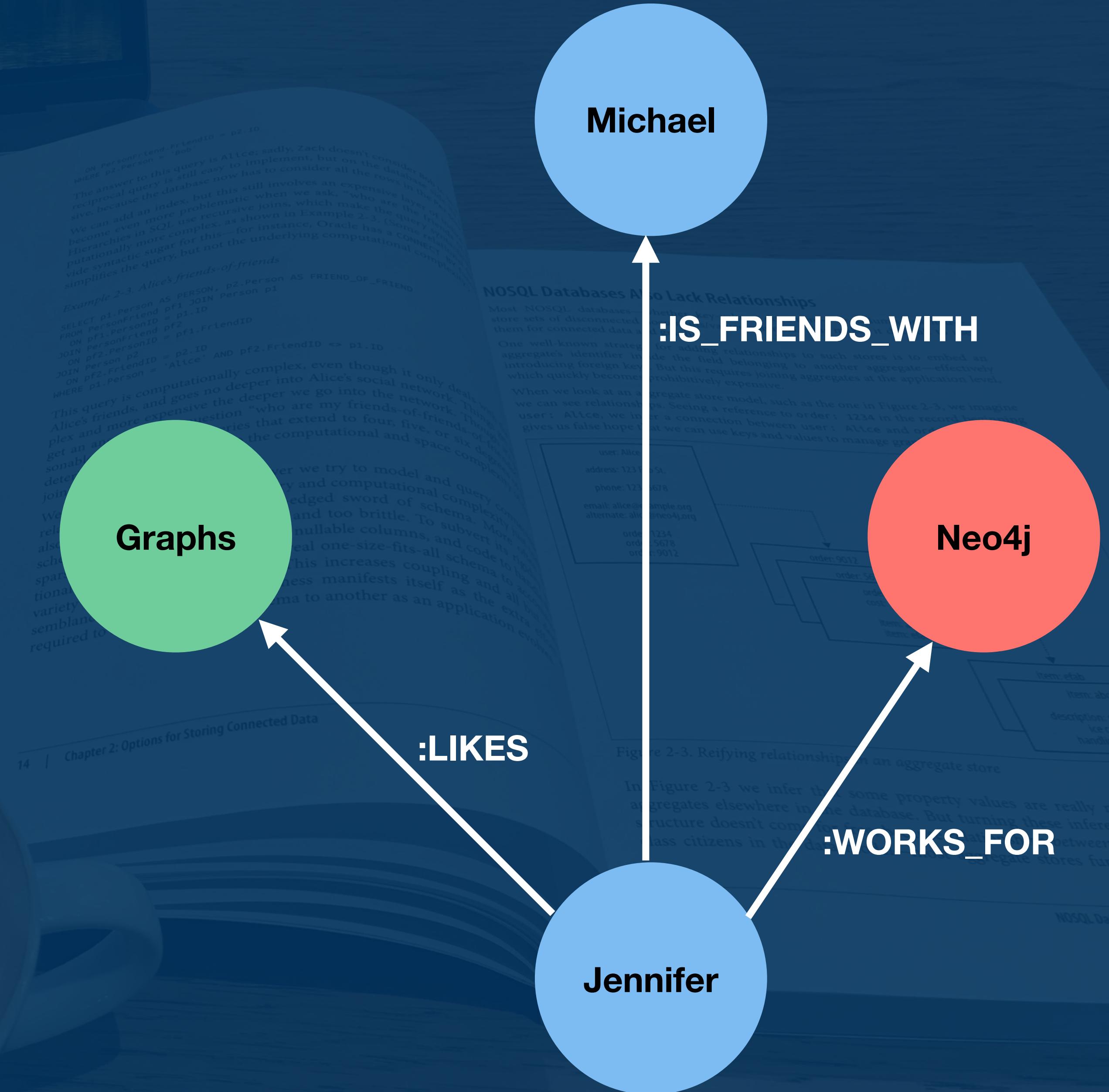
# Michael

# Neo4j

# Jennifer

# Graph components

- Node (vertex) Objects
- Relationship (edge) Connections



# Developing applications

14 | Chapter 2: Options for Storing Connected Data

```
+ Add another card  
on personFriend WHERE pf1.FriendID = p2.ID  
WHERE p2.person = 'Bob'  
  
The answer to this query is Alice; sadly, Zach doesn't consider himself reciprocated. The reciprocal query is still easy to implement, but on the database now has to consider all the rows in the PersonFriend table. We can add an index, but this still involves an expensive query. Friends-of-friends hierarchies in SQL use recursive joins, which makes the query exponentially more complex. Oracle has a CONNECT BY clause that provides syntactic sugar for this—for instance, Oracle has a CONNECT BY clause that simplifies the query, but not the underlying computational complexity.
```

```
Example 2-3. Alice's friends-of-friends  
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND  
FROM PersonFriend pf1 JOIN Person p1  
ON pf1.PersonID = p1.ID  
JOIN PersonFriend pf2  
ON pf2.PersonID = pf1.FriendID  
JOIN Person p2  
ON pf2.FriendID = p2.ID  
WHERE p1.person = 'Alice' AND pf2.FriendID => p1.ID
```

This query is computationally complex, even though it only deals with Alice's friends, and goes no deeper into Alice's social network. Things get an answer to the question "who are my friends-of-friends," but the computation becomes increasingly complex over time, queries that extend to four, five, or six degrees of separation. This is due to the computational and space complexities of joining tables.

## NOSQL Databases Also Lack Relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

When we look at an aggregate store model, such as the one in Figure 2-3, we imagine we can see relationships. Seeing a reference to order: 1234 in the record beginning user: Alice, we infer a connection between user: Alice and order: 1234. This gives us false hope that we can use keys and values to manage graphs.



Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really references to aggregates elsewhere in the database. But turning these inferred structure doesn't come for free, because relationships between class citizens in the data model—most aggregate stores fur-

# Language drivers

- Neo4j Java driver (official)
- Spring Data Neo4j
  - Uses Neo4j Java driver at base
- Other options:
  - Official drivers-> Python, JavaScript, Go, .NET
  - Java frameworks-> Quarkus, Helidon, Micronaut, etc

The answer to this query is Alice; sadly, Zach doesn't consider himself reciprocated, because the query is still easy to implement, but on the database now has to consider all the rows in the PersonFriend table. We can add an index, but this still involves an expensive query, which becomes even more problematic when we ask, "Who are my friends-of-friends?" Hierarchies in SQL use recursive joins, which makes the query exponentially more complex, as shown in Example 2-3. Some relational databases provide syntactic sugar for this—for instance, Oracle has a CONNECT BY clause that simplifies the query, but not the underlying computational complexity.

*Example 2-3. Alice's friends-of-friends*

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID
JOIN Person p2
ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID => p1.ID
```

This query is computationally complex, even though it only deals with Alice's friends, and goes no deeper into Alice's social network. Things get even more complex and expensive the deeper we go into the network. Thus, after a reasonable period of time, queries that extend to four, five, or six degrees of separation significantly due to the computational and space complexities of joining tables.

We work against the grain whenever we try to model and query connected data in a relational database. Besides the query and computational complexity, we also have to deal with the double-edged sword of schema. More often than not, schema proves to be both too rigid and too brittle. To subvert its rigidity in computational cases—all because there's no real one-size-fits-all schema to handle the variety in the data we encounter. This increases coupling and all but eliminates the semblance of cohesion. Its brittleness manifests itself as the extra effort required to migrate from one schema to another as an application evolves.

## NOSQL Databases Also Lack Relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

When we look at an aggregate store model, such as the one in Figure 2-3, we imagine we can see relationships. Seeing a reference to `order: 1234` in the record beginning `user: Alice`, we infer a connection between `user: Alice` and `order: 1234`. This gives us false hope that we can use keys and values to manage graphs.

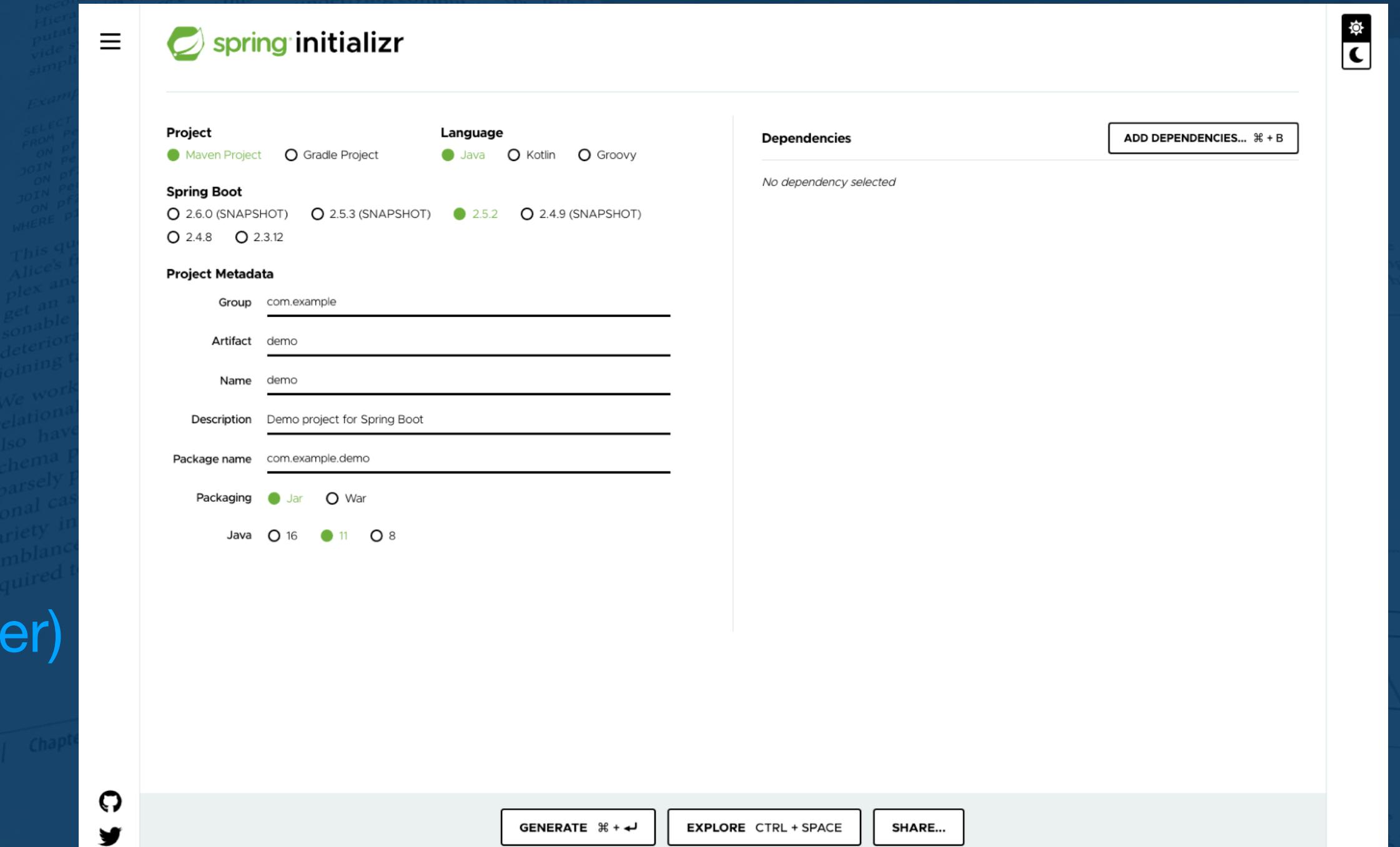


Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really references to aggregates elsewhere in the database. But turning these inferred relationships into actual connections comes at a cost. The structure doesn't come for free, because relationships between class citizens in the data model—most aggregate stores further

# Spring provisions

- Spring Initializr
- Spring Boot's reduction of boilerplate
- Annotation-based OGM for POJOs
- Query derivation + DSL for custom (Cypher)
- Spring Data's easy connection to Neo4j
  - BOLT



# Let's see some code!

Demo time!

14 | Chapter 2: Options for Storing Connected Data

```
    ON personFriend.FriendID = p2.ID  
    WHERE p2.person = 'Bob'
```

The answer to this query is Alice; sadly, Zach doesn't consider himself reciprocal. The query is still easy to implement, but on the database now has to consider all the rows in the PersonFriend table. We can add an index, but this still involves an expensive layer of memory. Hierarchies in SQL use recursive joins, which makes the query significantly more complex, as shown in Example 2-3. Some relational databases provide syntactic sugar for this—for instance, Oracle has a CONNECT BY clause that simplifies the query, but not the underlying computational complexity.

*Example 2-3. Alice's friends-of-friends*

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND  
FROM PersonFriend pf1 JOIN Person p1  
ON pf1.PersonID = p1.ID  
JOIN PersonFriend pf2  
ON pf2.PersonID = pf1.FriendID  
JOIN Person p2  
ON pf2.FriendID = p2.ID  
WHERE p1.person = 'Alice' AND pf2.FriendID => p1.ID
```

This query is computationally complex, even though it only deals with Alice's friends, and goes no deeper into Alice's social network. Things get an answer to the question "who are my friends-of-friends" in a reasonable period of time, queries that extend to four, five, or six degrees deteriorate significantly due to the computational and space complexities of joining tables.

## NOSQL Databases Also Lack Relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

When we look at an aggregate store model, such as the one in Figure 2-3, we imagine we can see relationships. Seeing a reference to `order: 1234` in the record beginning `user: Alice`, we infer a connection between `user: Alice` and `order: 1234`. This gives us false hope that we can use keys and values to manage graphs.



Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really references to aggregates elsewhere in the database. But turning these inferred structure doesn't come for free, because relationships between class citizens in the data model—most aggregate stores fur-

# Resources

- Source code: [github.com/JMHReif/pouring-coffee-into-matrix-lombok](https://github.com/JMHReif/pouring-coffee-into-matrix-lombok)
- Spring Data Neo4j documentation: [dev.neo4j.com/sdn-docs](https://dev.neo4j.com/sdn-docs)
- SDN reactive: [github.com/JMHReif/sdnrx-marvel-basic](https://github.com/JMHReif/sdnrx-marvel-basic)
- Neo4j AuraDB: [dev.neo4j.com/aura](https://dev.neo4j.com/aura) (FREE instance!)

Jennifer Reif

Email: [jennifer.reif@neo4j.com](mailto:jennifer.reif@neo4j.com)

Twitter: [@JMHReif](#)

LinkedIn: [linkedin.com/in/jmhreif](https://linkedin.com/in/jmhreif)

Github: [GitHub.com/JMHReif](https://GitHub.com/JMHReif)

Website: [jmhreif.com](http://jmhreif.com)

Figure 2-3. Reifying relationships in an aggregate store

In Figure 2-3 we infer that some property values are really references to aggregates elsewhere in the database. But turning these inferred relationships into real relationships doesn't come for free, because relationships between class citizens in the data model—most aggregate stores further