* What were the decisions taken in the modelling?
* Why were these decisions taken?
* What were the consequences of these decisions?

In the BookStore database that have been implemented in Neo4j, there are 7 different kinds of nodes and 7 different kinds of relationships that connect these nodes together.

Most of the nodes has a unique auto created UUID instead of using the normal id that Neo4j is auto generating it.

**Node IDs have a semantic and give the offset of that node or relationship within the store file. Consider the following example: Let’s say when deleting a node that has a reference in MongoDB (or any other third-party database), and if we forget the reference in Mongo to that now-deleted node.  
  
After the original node is deleted, there’s a free space in the node file, so when creating a new (likely unrelated) node, it now uses the same previously-used Node ID. The dangling reference now points to something semantically completely different, which can cause huge problems in database.  
  
So instead of relying on the semantic-driven ID, it is possible and better to store** [**UUID**](http://searchsoa.techtarget.com/definition/UUID) **property , place an index on that and then reference the UUID. Then, even when deleting the node in the future, and forgetting about the reference from a third-party system, it will end up with an error message instead of a dangling reference.**

The Customer is a node which stores the following information about customers:

name, email, password, customerNo as UUID is connected to two different nodes using two kinds of relationships.

It has Lives\_In relationship connecting it to the Address node which stores street, zip, and town for each customer.

The decision of having the address as a node and not having it embedded in the customer node is to avoid redundancy in the database. For sure Queries have been taken into consideration as well,

For instance, if a customer has changed his address it is also possible to execute this process in an easy way if the address is a separate node.

The customer node has another relationship called PURCHASED to the order node which stores only date and orderNo as UUID, while the other properties that must be contained in an order (price, qty, and ISBN of an book) are stored in the relationship PURCHASED between the customer and the order nodes.

So, it is easier to check how many books a customer has bought in a single order by counting the number of purchased relationships between a specific customer and a specific order.

In addition it is easier to check the number of books a customer has bought in a single order through the graph since simply the number of purchased relationships is the number of books.

The decision of having the price in an order and not having a reference to it in the Book node which stores all the information about books including the price, is that a book may be on a discount or a different price when it has been ordered by an customer and when this customer wants to return the book so he will get back the same amount of money as when he bought the book.

Using a relationship called CONTAINED the order node is connected to the Book node, which is the main and the most important node, which stores information about books, each book has ISBN, pages, copies, price, currency, language and title.

Author is a separate node, as it is an important node and a lot of information about authors can be necessary to store, name, email and UUID, the author node is connected to the book node using WROTE relationship.

A book can have more than one author so having it as separate node avoid redundancy.

In the same way a separate node has been created to store Characters of a book which in this case has only one attributes called type.

The character node in connected to the book node using PRESENTED\_IN relationship.

A book has one more relationship called IS\_a which connect the book node with the Category node which stores only the name of a category.

This node is recursive, it has a relationship of type PARENT to itself.

* What were the difficult and easy parts of the exercise?

the most time-consuming queries in the exercise was everything related to the recursive node (Category), it just took some more time to understand how to execute queries on a recursive node.

It was also a bit strange that group by is not provided on everything in Neo4j as it is only possible to do group by on aggregate functions, so all the exercised that was possible to solve them using group by in relational database and Mongodb has been solved in a different way here in Neo4j.

On the other hand it was interesting to play around the data in Neo4j and execute queries and see results in colourful graphs beside tables.