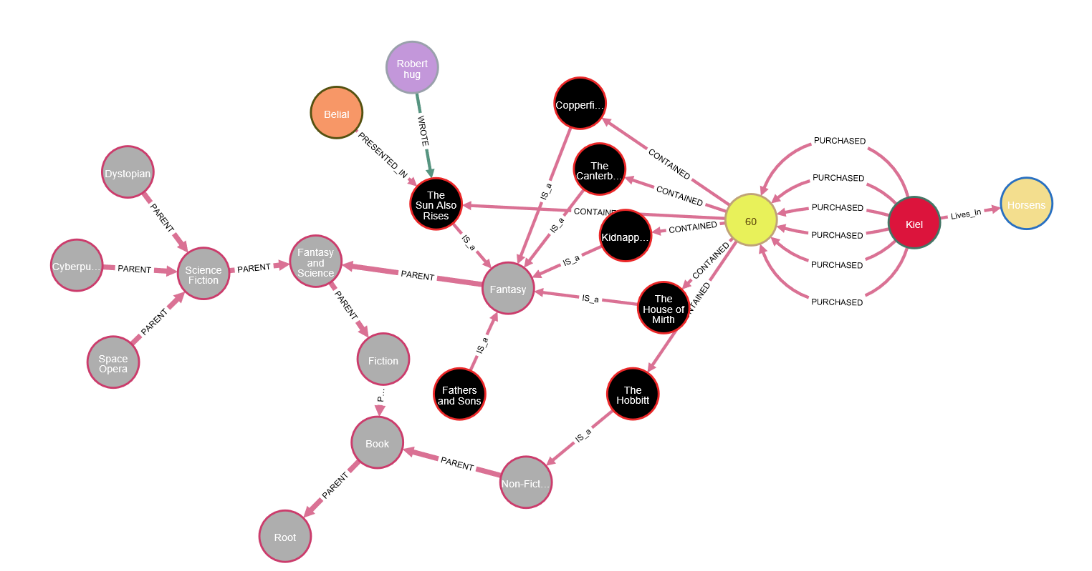
***IMPLEMENTING THE BOOKSTORE DATABASE IN NEO4J***



 In the Bookstore database graph above that has been implemented in Neo4j, there are 7 different kinds of nodes and 7 different kinds of relationships (Edges) that connect these nodes together. Instead of using the normal id identifier, an auto created UUID property has been used in some nodes. That can help in the following scenario: let’s consider that each node or relationship in Neo4j has a unique id, and assuming this id has a reference in a third party database for instance MongoDB, then when deleting this node or relationship and forgetting about the reference it has in the third party database, that will leave a free space in the database that can be filled by any other new generated node that could be completely different from the deleted one, that means that the reference in the third party database is pointing to the wrong value now. That is of course not convenient and could cause huge problems in the database. So, by using UUID instead of id, when the previous scenario happens, in this case the reference in the third-party database will return an error instead of returning a wrong result.

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 on 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 even 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 and retrieve later (name and email...)  the author node is connected to the book node using **WROTE** relationship.

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

In the same way a separate node has been created to store **Character**s of a book which in this case has only one attribute called type.

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

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

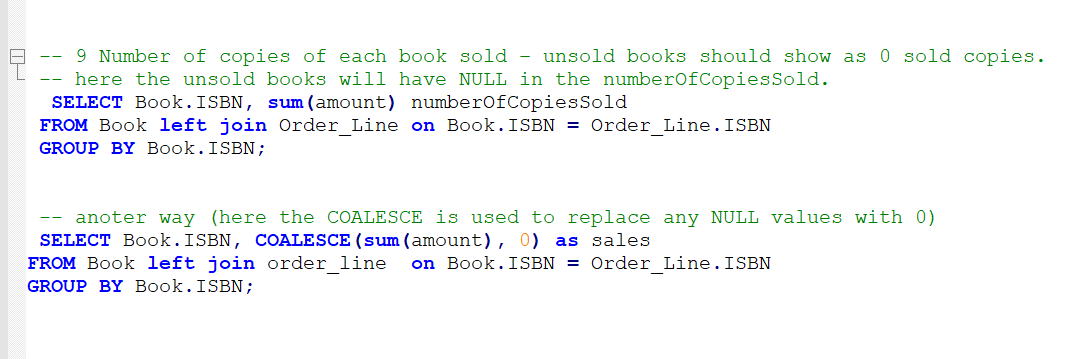
This node is recursive, it has a relationship of type PARENT to itself.

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 in Neo4j.

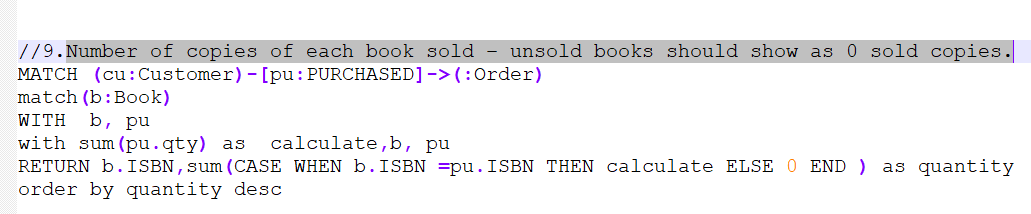
It was also a bit strange that group by is not provided on everything in Neo4j as it is only possible to do group by on aggregate functions, so all the exercises that was possible to solve them using group by in relational database and Mongodb has been solved in a different way here in Neo4j.

For Instance, in the exercise number 9: return number of copies of each book sold – unsold books should show as 0 sold copies.

In the relational database it is possible to solve it by doing a left join between the book table and the order table, and group by the book's ISBN, as it is shown in the script below.



But in Neo4j it is not possible to do joins and it is not possible to group by any key without using an aggregate function as sum, the script below shows how the exercise has been solved 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 bedside tables.

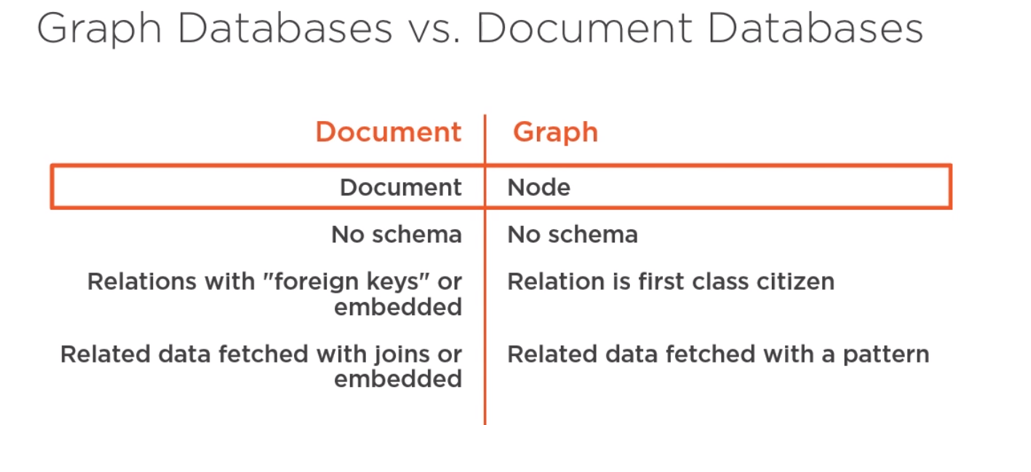
\_\_\_\_\_\_\_\_\_\_

In the world of databases, it is essential to find the right tools for the right job. Thus, the “one size fits all” proposition of large relational systems was replaced by conscious decisions about choosing the tools that fit each concrete use case.

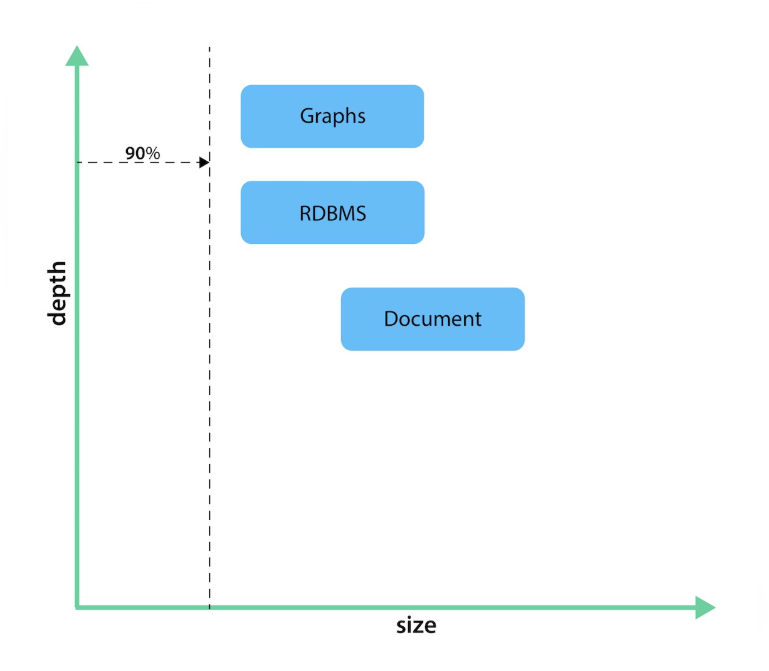
In this section we are going to compare graph database, neo4j in particular, to its sibling namely MongoDB.

Mongo is an aggregate oriented database system, which means that it groups the data based on criterion (document that has key-value pairs) this approach forms one dedicated view of the data. The problem in this approach comes to the view when we want to connect multiple documents together, or when we have a graph structure where all the documents are connected. There is a work around this, since we can embed documents inside other documents, but the trade of in this case will be redundant data. Another possible solution to highly related document in Mongo is to reference by foreign keys. By referencing instead of embedding, we can embed a document’s identifier, typically an id, that references another document. However, this approach is prohibitively expensive if we want to join documents at the application level most of the time. In other words, the more joins we have the slower the query will perform. Indexing the foreign key will speed up the process in this case by making the aggregation perform faster. The problem introduced here is important when we want to modify the data frequently. To protect referential integrity when modifying the data, we must implement transactions manually. Moreover, we must configure the database to block any access to the data until the update is done.

Neo4j supports complex related data with as much depth as needed and the update scenario is much convenient since neo4j supports transactions that conform to ACID standards. One thing to point out though, the transaction is not supported through the bowser application, it requires an implementation of the Driver provided from neo4j. using the driver means the application will call neo4J REST API or Bolt by opening a session with the transaction manager and doing whatever job we want to do. The transaction session will will manage the successful scenario by either update all the nodes involved in the transaction or it will roll back the changes. It is still possible to achieve atomic queries in the browser though, by combining all the queries in one single statement.



The figure below shows the key differences between RDBMS, Mongo, and Neo4j :



First, Mongodb can handle big size data and it is very good at that specially when it comes to multimedia files. The downside comes to the picture when we have a depth in the data such that it must link many documents either with referencing or documenting.

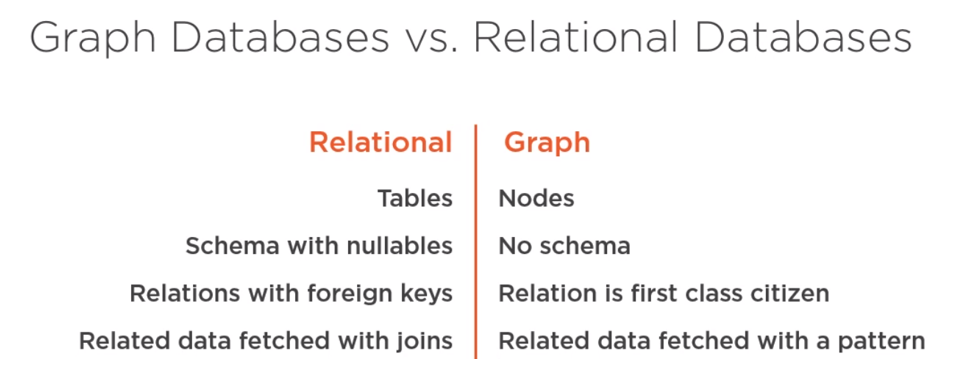
Another point for mongo is its flexibility regarding the schema less model. Even when we apply a restricted schema to mongo, it is still considered more flexible than SQL since it can still handle the data with some extra attribute. Mongo, however, is bad at linking documents together because as described in the first part of this document, it needs to perform many aggregations to view the data in the desired form. Moreover, mongo has a limit of embedding documents inside other documents, which might be an issue if we go with embedding instead of referencing.

the graph paradigm to store data:  neo4j is very strong in storing and retrieving highly related data, and the developer can still benefit from the schema less feature. even though there are many relationships between the data, a graph database will remain very performant during data retrieval, even with millions of nodes. That means we can add or delete nodes and property of nodes without affecting already stored nodes. Because of that, a graph database responds well in an agile environment where changes over the data are very common

Neo4j, is very good in data that requires many links, data with a high depth, it is faster than SQL and Mongo in performing queries against across documents/tables/nodes.

RDBMS on the other hand, can handle data with a high depth, but its performance is slower than neo4j. RDBMS is not the best choice for large data, large text multimedia data with unknown shape since the schema in RDBMS cannot handle undefined attributes.

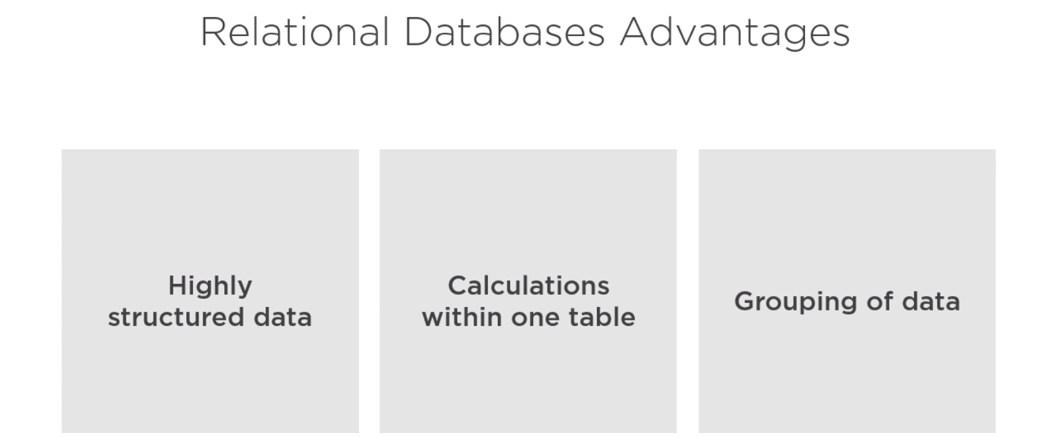
General comparison between RDBMS and Graph



RDBMS has a predefined schema, and all the data must match the schema exactly, when we have a record that has columns with fields that are not applicable, we can mark them as nullable values in the schema. Thus, the application must check for null values and handle exceptions in such a case. Graph on the other hand, consists of nodes instead of tables, and each node is on its own, which means that we do not have to work with null values, just leave out or add properties when needed. In RDBMS, the foreign key paradigm approach is applied to link tables. To get data from a related table, we must perform a join query. In case the data is spread among multiple tables we need to perform multiple joins. The more joins there are in a query, the more the performance suffers exponentially

Graph databases have relationships besides the nodes, the relationships are first-class citizens of the database, which directionally point from one node to another node. Relationships in a graph database are just as important as nodes and are separate from them. Therefore, there is no data duplication, and the cypher query is optimized for relationships too, the query targets a pattern that is matched by the database. Querying nodes related to other nodes is easy and take more time to execute, but the increase in wait time does not compare to a multi-joined in relational databases

When to choose a Relational database:



1 Relational databases will beat a graph database if, this example is from the book store assignment, a sum of the total price for an order has to be calculated or the average age of customers.

 Relational databases are not very good at retrieving nested data.

Example from the assignment: the foreign key CustomerId in the order table points to a primary key CustomerId in a customer's table. The foreign key orderNo and bookISBN points to order and a book. We need several complicated and costly joins just to find out what book a customer bought. But it gets even more complex if you want to ask the database which customers bought a certain book, and to find out which customers bought this book who also bought that book, like Amazon does for finding customer trends.

RDSMS is not good at highly related data

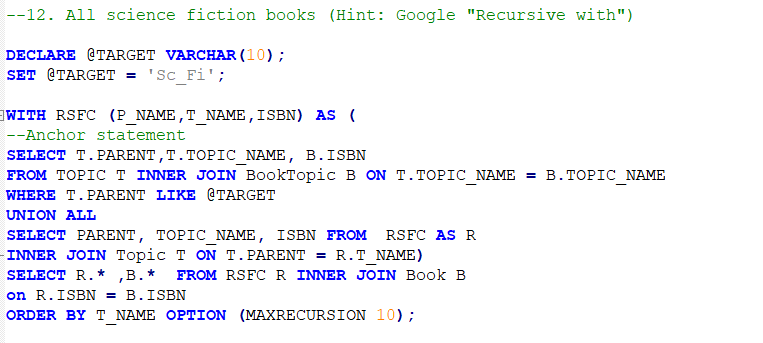
Example from Facebook:

If we want to find friends of friends at minimum level 5 depth and compare the performance between SQL and neo4j.this Query will break SQL off because it just took too long, when Neo4j produced the results much faster and much simpler.

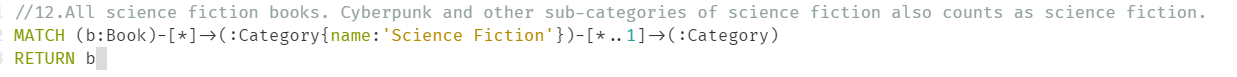
The complexity of recursive queries in SQL vs neo4j

Question 12 is an example of how recursive queries looks like in neo4j vs SQL

SQL



Neo4j



Where in neo4j this is enough to do the same job, with no need of any temporary table to save the result from each level and then join it with the next and so on.

The key difference between mongodb and neo4j for this exercise is managing transactions. In Mongo transaction is done using many helper functions and complicated procedures to get the job done in an ACID manner. Since mongo update works on the document level, it does not support atomic update across multiple documents. In neo4j this scenario was manageable with less effort since we could write multiple commands in the same query and if one fails none of them will be applied.

To be fair with this, managing rollbacks in neo4j was so difficult and we could not figure out whether it can be done in the browser or not.

The writebulk function in mongodb supports multiple statements within one scope, but it does not support rollback in case if one of the queries fails.

|  |  |  |  |
| --- | --- | --- | --- |
|  | SQL | MongoDB | Neo4j |
| ACID | Follows the ACID (Atomic, Consistent, Isolated and durable). This means that once a transaction is completed, the data remains consistent and stable. | Follows the **BASE (Basic Availability, Soft-state and Eventual consistency)**model.  It supports **atomic updates on a single document level.**  MongoDB supports atomic modifications. Meaning, when updating two values within a document, either all two are updated successfully or remains unchanged.  MongoDB provides tenable consistency model through the readConcern and writeConcern parameters. | supports properties of ACID:  * READ\_COMMITTED is the default isolation level. * Non-repeated reads could happen (example: write locks are only acquired and held until the end point of a transaction). * Write locks can be acquired manually on relationships and nodes to acquire higher isolation level. * Locks can be acquired only at the node and relationship level. |
| CAP TheoremA picture containing text, map  Description automatically generated | It opts for **C**onsistency and **A**vailability (**CA**). This means that data will be consistent between all nodes as long as nodes are online.  This will also allow developer to read/write from any node and be sure that the data is consistent. If ever a partition between node is developed, the data will be out of sync and won’t resolve until the partition is resolved. | opts for **C**onsistency and **P**artition tolerance (**CP**). This means that the consistent view of the database will be available for all the clients to see.  Though the users of one node will have to wait for any other nodes to come to an agreement before being able to read or write to the database. In this case**, the availability takes a backseat to consistency.** | opts for ****C****onsistency and ****A****vailability (****CA****).high consistency and availability with no partition tolerance as most of the traditional RDBMS |

## SQL /NoSQL Comparison

Summary:

Graph databases are flexible and performant with highly related data

Relational databases are great for reporting and transactional queries

Document databases are great to store object

All databases have their place, and it always depends on the use case to select the right tools

The best implementation of this exercise will be in SQL since there is not much depth level in the data, and the shape of the data is well known in advance. Moreover, managing transactions in NoSQL databases is not optimal yet.