MongoDB LAB - CBDE ***07/12/20*** *Pau Canosa - Laia Ondoño*

QUERY 1

In this query, we've noticed that all the attributes used belong to the LineItem class. Thus, we've just added to our design class LineItem. In particular, the data from LineItem will remain inside the Order collection. This decision will be explained later in this document. Anyhow, query 1 will retrieve the data from the lineitems inside the order's documents.

In order to boost the performance of this query, we will use an index into *l_shipdate*, since it is the only condition written.

QUERY 2

This query is quite more complex than the first one. It combines 5 classes: PartSupp, Supplier, Nation, Region and Part. The clue in this case is to understand the flow between all of them. From PartSupp you can navigate to Part and Supplier using *partKey* and *suppKey* respectively. Plus, from Supplier you can reach Nation and Region.
This means, in fact, that you can retrieve all the information from one single class: PartSupp.

Thus, the optimal design in this case is to have a single collection, PartSupp, which also contains the data from the other classes (Part, Supp, Nation and Region).

Again, in order to boost the performance of the query we will use an index in *r_name.* The criteria used to determine this index is based on the selectivity and the memory usage ensuring that the query will have an optimal performance.

QUERY 3

This query requires you to make important decisions regarding the database design. It involves the classes Order, Customer and LineItem. At this point there are many valid approaches to use, but we believe that there's one which gives a better performance.

The main issue here is how to handle LineItem and its relation with orders and customers. We could consider moving Order and Customer into the LineItem class from the query one, but in terms of scalability this approach fails: If we take a look at the lab instructions, in particular at the TPC-H schema, we find that LineItem has a cardinality of SF*6.000.000. Although the cardinality for the LineItem table is approximate, it's the biggest among all classes. Adding Orders and Customer into LineItem would mean that these two classes would have the same cardinality, and there would be plenty of documents with the same information (i.e an order of 200 products would result in 200 line items documents with the same order and customer information for every tuple).

In order to avoid as much as possible the redundant data, we find having a new collection the best solution. However, there are two options to consider.

Firstly, we could have Order as the main class with the customer information and just the needed information from LineItem for the query, and another collection LineItem, independent of Order, which we would use for the query one.

Although this solution would improve in scalability from the first approach, and less redundant data would be involved, there is one last approach that we find optimal.

Our solution lies in having Order as the main class, with the customer information and the respective LineItem information inside as an array of items. This solution would improve in scalability from both approaches and would reduce almost all the redundant data assumed before. With this design, we keep the LineItem information just once. Another benefit from this approach are the updates, since the orderKey is one of the primary keys from LineItem, which means that the cost to find the document where the LineItem belongs won't be expensive.

We recognize that our solution implies a certain amount of data which is redundant (some Nation, Region and Supp data, we will explain later) but we believe that we can assume this extra “information” if it’s coupled with an optimal performance in terms of time.

We will use an index among the *c_mktsegment* in order to boost the query performance in terms of time. We use this index because it is the one that belongs to the class with lower cardinality.

QUERY 4

The last query combines all the classes we’ve talked about in the previous page. If we take a deep look at the query we will realize that in the end is quite similar with the third one, but in this case, we need information from Supplier, Nation, and Region.

Again, we are dealing with a similar problem as the one in the third query. And the main purpose in this case is to solve the question: is it better to add the needed information to the Order collection explained before, or is it possible to redesign the database in such a way that we avoid having references and redundant data and we keep an optimal performance?

From our point of view, we feel that keeping the Order collection is the right choice. We will keep the information regarding Supplier, Nation, and Region in each document from the collection. We prefer having some redundant data (in this case it’s not much of a deal) and gain performance in time, without the usage of references.

In this case, the index on *r_name* will also be useful.

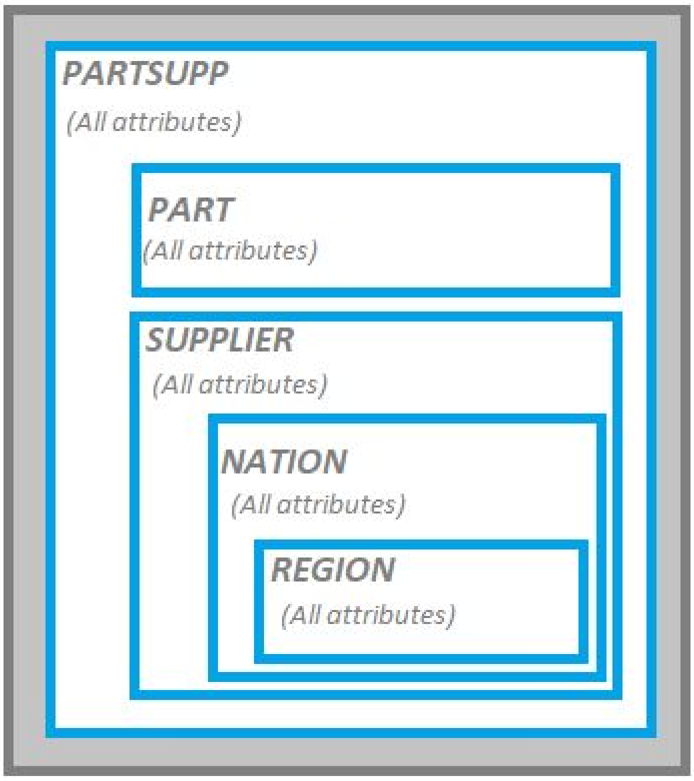
CONCLUSION

Firstly, in order to make our program more readable and understandable, through the inserts section you will find that we have included all the attributes from each table. This means that there is more data than the necessary to retrieve the query results. We are aware that the amount of data necessary is smaller.

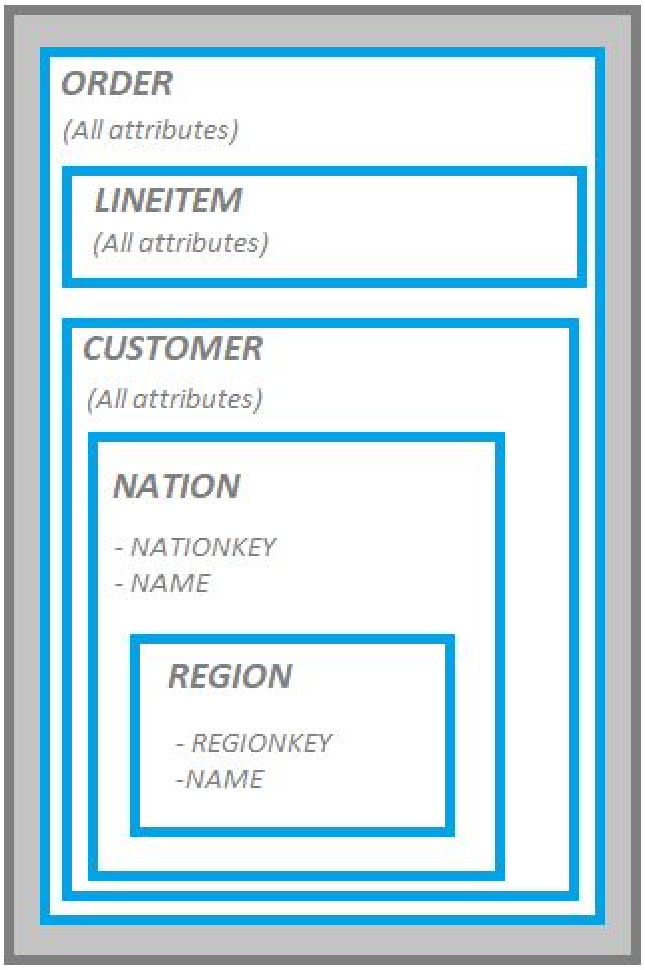
In our Python program you will be able to visualize the actual dataset and execute any query. The dependences involved in the program are *pymongo, datetime* and *random.*

To sum up, and with the purpose of making our design easier to understand, we attach the structure of each collection involved.

(QUERY 2)



(QUERY 1 - 3 - 4)


ORDER
(All attributes)
LINEITEM
(All attributes)
CUSTOMER
(All attributes)
NATION
- NATIONKEY
- NAME
REGION
- REGIONKEY
-NAME