Synopsis

Introduction / Motivation

The topic for this synopsis is NoSQL databases. We will look into how the NoSQL MongoDB can be used in a database context and look at the pros and cons of using this database for a specific example. We will then compare this setup with the relational database Microsoft SQL Server, referred to as SQL in the following.

We will consider the following example: We are starting an online web shop. In the startup phase we want to keep things simple. Thus, initially we only want the following data stored:

- Customer data: Full name, email, and address, credit card information
- Inventory: item name, items in stock, timestamp of inventory updates and changes, price and category
- Order data: Timestamp of purchase, items ordered and their price, and the total cost of the order.
- Logging data: Error messages regarding order transactions and server connection.

After 1year the business is successful, and we want to add the following to our data storage:

- Customer data: mobile nr and birthday
- Order data: Discount code
- Logging data: session login start and end, items clicked, time spent on each item page, purchased items.

This allows us to increase customer service by enabling SMS service when packages have been delivered. Further, the added logging data allows us to understand our customers better as we can now start analyzing their behavior on our web shop.

This example is representative for many types of startup companies, which needs to store some kind of data, and some of the considerations they need to be aware of depending on their vision for their company – or considerations needed by some data service provider. However, the same considerations are relevant for established companies on an ongoing basis in order to either adapt or be at the forefront of their (changing) data needs.

I have chosen this problem as data storage is at the core of data management. Poorly or ineffective storage of data will affect all parts of a business such as data scaling, data access and readability, and data consistency and trustworthiness.

Problem Statement

Our use case is the example described above, and it is a high priority for us that the stored data is accurate and trustworthy. Further, we want order data and logging to be easily available for analysis, where logging data must also be easily adaptive for ongoing changes.

With these main criteria our focus will be on answering the questions:

- Is MongoDB the better choice for the type of data we want to store?

- Is MongoDB able to handle scaling of the business and changing requirements to data storage needs?
- Is MongoDB preferred over SQL? And why or why not?
- Is a hybrid of databases the way to go – and why or why not?

Methodology

To answer above questions, we will start by considering the type of data we are handling, is it structured or non-structured. Is the data to be stored likely to change over time, and how often. We will then compare this with the pros and cons of MongoDB – keeping our data criteria in mind.

We will give an example of how to make a minimalistic implementation of our use case in MongoDB and SQL, and how to handle the migrations when changing the data storage needs after 1year. We will compare the two database setups and some of the considerations that needs to be done and up front decision making. We will look at the pros and cons of storing the data in each database type, also keeping potential scaling in mind. Based on this we will argue which type of database choice is better suited for this specific use case based on its criteria. The database choice(s) should be the one that best fulfills the criteria of this use case.

Note, in the minimalistic implementation, credit card information will not be included. If we were to include this, common practice would be to store this information separately and encrypted. Further, logging data will not be implemented in SQL.

Literature references

Lecture material: W37 T-SQL & Stored Procedures, W46 Concurrency Control & Transaction Management

MongoDB Schema Design: Data Modeling Best Practices
https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

Cap Theorem: https://www.youtube.com/watch?v=BHqjEjzAicA&ab_channel=ByteByteGo

YouTube videos:

- SQL vs NoSQL or MySQL vs MongoDB
  https://www.youtube.com/watch?v=ZS_kXvOeQ5Y&ab_channel=Academind
- ACID Properties in Databases With Examples
  https://www.youtube.com/watch?v=GAe5oB742dw&ab_channel=ByteByteGo
- Acid Transactions in MongoDB
  https://www.youtube.com/watch?v=tkDcNltEh1s&ab_channel=Observer
- SQL vs. NoSQL Explained https://www.youtube.com/watch?v=_Ss42Vb1SU4&ab_channel=Exponent

Analysis & Results

First let us consider the nature of the data that needs to be stored. Customer, inventory and order data are structured data types, i.e. they fit a fixed format suited to tables with well-defined fields. Although changes or additions to the fields may occur, like in this use case after 1 year, they most likely won't occur very often.

Logging data on the other hand is classified as semi-structured data. We can introduce some structure using key-value pairs, but it is more flexible and does not suit a fixed schema. We need to impose some structure on this data in order easy our querying. Fields to be stored in the logging data may change more frequently based on ongoing changes in business needs, e.g. for analysis and error detection.

The semi-structured data is ideal to store in MongoDB, as MongoDB is a schema-less database where data is stored in BSON documents (binary JSON documents). This makes migrations, such as the addition of new fields, very easy as there is no fixed structure one needs to adhere to – unless we impose it ourselves. Therefore, the logging data is ideal to store in MongoDB. That being said, one needs to be aware that the BSON documents have a limit size of 16MB. One of course needs to bear this in mind if logging data for a customer login session is escalating and a vast majority of data needs to be logged. To mitigate the 16MB size limit one could either split the logging file into smaller files, all having the same session id, or use referencing to split the logging file. Further, depending on how much structure we impose in the logging data, this will affect query performance as unstructured data can be harder to query and analyze, as we may need to loop through multiple documents searching for relevant fields. When scaling, both data and business wise, the lack of structure will likely create large bottlenecks if left unhandled.

The structured nature of the customer, inventory and order data is very suitable for SQL. But it can of course still be stored in MongoDB. In this case, since data accuracy and consistency were a high priority for us, we need to impose and maintain the structure of the data. This can be one of the pitfalls of MongoDB, as due to its very flexible and schema-less design we can quickly setup and store data but given it too little though and initial planning, could result in a big mess later that can be very time consuming and costly to fix.

Another aspect needed to be considered in MongoDB is whether or when to use embedding or referencing, and how this may affect ACID compliance – which in our use case is a high priority to keep.

One of the advantages of embedding is that it enables very fast lookups. As example, in the order data, we can embed all relevant customer information. By embedding we get duplication of data but at the same time we avoid the need for looking up references in other documents. This would make order detail retrieval very fast and effective as all relevant information is stored together in the same document. As also mentioned in literature article[1] we should not avoid duplication, i.e. embedding, if it enhances performance based on frequent queries relevant for our use case.

When using embedding we should be aware of how this may affect ACID compliance. By default, if only a single document is updated it is ACID compliant. However, if multiple documents need to be updated, due to embedding, we can experience false reads or latency, depending on how documents are updated. As example, assume a customer changes his last name. This customer has made several purchases in our web shop, and in each order, we have embedded his customer information including his full name. Now we need to update his surname in all relevant order documents. This can be done in different ways. One way is to keep all order documents available for query during the update. Until the update is done this can result in false reads, e.g. when searching on orders using his new name all orders may not be shown as they may not have been updated yet which is a breach in ACID compliance. Another approach could be to lock all relevant documents until they have been updated. This corresponds to them not being available for query while the update is ongoing resulting in latency, on the other hand it ensures data consistency and ACID compliance.

This is basically a small-scale example of the CAP theorem. So how would we handle this in our use case if the business scaled. Assuming the business becomes a huge success and scales to e.g. an international

[1] MongoDB Schema Design: Data Modeling Best Practices | MongoDB

level. First of all, MongoDB is very agile in this regard, since if the data becomes too large to keep on one server it can be scaled horizontally by splitting data across multiple servers. This is called sharding. As example, data allocated to certain regions or a country could have a separate server. Just like the small-scale example with ACID compliance and embedding, we here need to be aware and make decisions based on tradeoffs according to the CAP theorem which states: when you have a distributed system, like we do when using MongoDB and sharding, you have to choose between the tradeoffs of:

1. Consistency: all nodes/servers have a consistent view of the data
2. Availability: ability to respond to a request/query at all times
3. Partition tolerance: ability of a system to continue operation if there is a network partition, i.e. a network failure where nodes/servers cannot communicate

So, if we experience a network partition, do we allow false reads from different servers and prioritize availability, or do we prioritize data consistency resulting in latency, i.e. data not being available. This is not only regarding order data but all our data. As example, do we prioritize consistency where customers will always see the correct quantities in stock at the cost of the web shop may be unavailable for the duration of the partition. Or do we prioritize availability where the web shop is up and running but may show incorrect stock levels. Note this could result in our stock levels becoming negative, having sold more than what is in stock, if customers get false reads with incorrect stock levels on some servers due to lack of the latest purchase updates. This would leave us with a data mess that needs to be cleaned up when all servers are back up and running. In practice we would not have to choose between either consistency or availability as we can have a combination of both to some extent. As example, in case of a partition, we could leave the web shop up and running where customers can add items to their shopping cart, but they cannot complete their purchase until all servers are consistent. Since one of our criteria for our use case was consistency, we would in this case, both in the small- and large-scale scenario, prioritize consistency over availability.

Based on the above, what does SQL have to offer compared to MongoDB. First of all, SQL is a relational database with built-in structural requirements. As mentioned earlier customer, inventory and order data is structured data by its nature and thus suitable for SQL. Depending on how much structure we choose to implement in MongoDB, more initial planning is needed in SQL when designing tables and accounting for potential future migrations, as illustrated in the appendix Figure 1. This initial planning may be well worth it as it ensures structure in our data. Migrations in SQL can be more cumbersome than in MongoDB, again depending on our self-imposed structure. Migrations are easily made ACID compliant in SQL by using transactions as illustrated in the appendix Figure 8-9. Note in our minimalistic SQL migration example, test of the migration has not been implemented nor has a rollback plan for the different migrations. These steps can be quite time consuming, but the extra effort will be very well worth it as it helps us ensure, as best as we can, data consistency and trustworthiness, and in case we will ever need to roll back any migrations and bring our system back to a given state this will always be possible. ACID compliant transactions are also possible in MongoDB, however the relational table structure of SQL can help us maintain a higher level of structure and leave less room for errors.

Further, due to the table structure and relations between tables, SQL is also very suitable for complex queries and analysis, as it effectively handles joining and filtering of tables. Complex query performance and speed may be preferable in SQL compared to MongoDB, depending on the complexity of the query and our imposed structure, if any, in MongoDB. Of course, MongoDB also allows for filtering and aggregational pipelines, but for more advanced queries SQL can be more effective, despite using indexing in MongoDB.

In regards to scaling of the business, SQL can handle this as well by vertical scaling where the resources of the SQL server is increased, e.g. by increasing the RAM and upgrading the CPU on the computer running the SQL server. This is a simpler approach and easier to implement compared to sharding with MongoDB, keeping in mind that there will be an upper limit of vertical scaling compared to horizontal scaling.

So, based on our use case and our criteria, is MongoDB the better choice compared to SQL for our data? Since data accuracy and trustworthiness is a high priority in our use case, we can argue that SQL would be a better choice for storage of customer and inventory data due to its inherent structure well suited for this type of data. The "downside" is that more initial schema planning may be needed compared to using MongoDB. This "downside" however should be considered as a positive thing, as we need to think ahead and consider what type of data is of relevance for this use case, and what other types of data that may need to be stored further down the road or within the foreseeable future. These considerations are important already in the startup phase as they can save us much time later on.

Regarding order data, which is also structured, we can consider different angles and select which one is the most useful for our use case. As illustrated in the appendix Figure 4-5, order data can be split into two tables in SQL called orders and orderItems. This split is made to minimize redundance of duplicated rows and fields, but in our implementation some duplication is still present, which is okay. Storing the order data in this way in SQL results in a complete order not being available unless we query it ourselves. The logic for doing this could be stored in a stored procedure in SQL, as illustrated in appendix Figure 6-7. Of course, the stored procedure would need to be updated if we implemented migrations affecting the relevant tables. On the other hand, storing the order data in MongoDB and embedding the customer information would lead to very fast lookups of the complete orders, as all relevant order information would be stored in the same document. Again, as data consistency is a high priority in our use case I would argue for the pros of storing a complete copy of the entire order including all relevant information in MongoDB. By imposing some structure on the orders, they are still easily queried for analysis. Further, as the completion of an order will update the inventory stored in SQL, order analysis could still be performed in SQL via the inventory table – and if customer information is of relevance for the analysis the customer id can be extracted from the order stored in MongoDB.

The logging data, due to being semi-structed, is best suited for the more agile and schema-less design of MongoDB as SQL is suited for structured data.

Thus, to sum up, due to data accuracy and trustworthiness being a high priority in our use case, MongoDB is not preferred over SQL when considering the customer and inventory data. Despite migrations in SQL being potentially quite cumbersome, this is prioritized and considered as a positive as migrations is done with transactions, keeping data ACID compliant, and consistency and trustworthiness of our data remains high due to migration tests and tested rollback plan.

However, for storing the order and logging data we do find MongoDB to be the better choice, where MongoDB is very capable of handling changing requirements of our data storage needs, e.g. such as ongoing changes or addition of fields to the logging data. Further, MongoDB can handle large scaling of the business via sharding. In the expanding business case and using sharding in MongoDB, in our use case consistency is prioritized over availability according to the CAP theorem. For the order data we would use embedding to include all relevant order information and thereby storing an exact copy of the entire order.

In our use case a hybrid of the SQL and MongoDB databases would be the better solution, storing customer and inventory data in SQL, and order and logging data in MongoDB. For analysis purposes, we would need to impose some structure to the data stored in MongoDB. However, such structure would be implemented

per default in the construction of the web shop as inventory updates, purchases and orders would be set up to be handled and stored automatically when a purchase is completed. We have not implemented this functionality. Note that for improved customer experience, data such as inventory levels and items in the shopping cart, before a purchase is completed, would be stored in Redis cache, another type of NoSQL database which we will not look further into here.

Conclusion

We have considered a concreate use case and its criterias. Based on this we have looked into the NoSQL MongoDB from different perspectives in order to determine whether MongoDB was a suitable database to store our use case data in compared to the relational SQL database. As there is not a “one-solution-fits-all” model we can use, we had to base our data storage choices based on the specific use case and aspects of importance of this use case.

Considering the pros and cons of the two database types, the type of data considered and use case criterias we concluded that a hybrid database solution would be the best fit for this use case instead of only using MongoDB. The hybrid solution does impose a potential increase in workload up front as storing data in SQL does require some initial and forward thinking due to the structured table schema design and (potential) future migrations, which is more rigid compared to MongoDB. However, this exercise is strongly recommended and should be done also if only using MongoDB or any other NoSQL database, as it may prevent issues further down the road, in case of a potential business scaling or general changes in data requirement needs.

Appendix

In the practical part we have implemented customer, inventory and order data in both SQL and MongoDB. Logging data has only been implemented in MongoDB. This is to illustrate the different workloads based on the choice of database. However as mentioned previously in the Analysis & Results section, the additional workload when using SQL can be considered as a positive as its rigid structure helps enforcing data consistency and trustworthiness, for structured data and thereby leaves less room for errors.

Figure 1 shows an example of the initial workload that needs to be done before being able to implement the customer, inventory and order data in SQL, where order data in this example is split into orders and orderItems to minimize redundancy. Note that we concluded order data should be stored in MongoDB. This is just to illustrate how it could have been done in SQL.

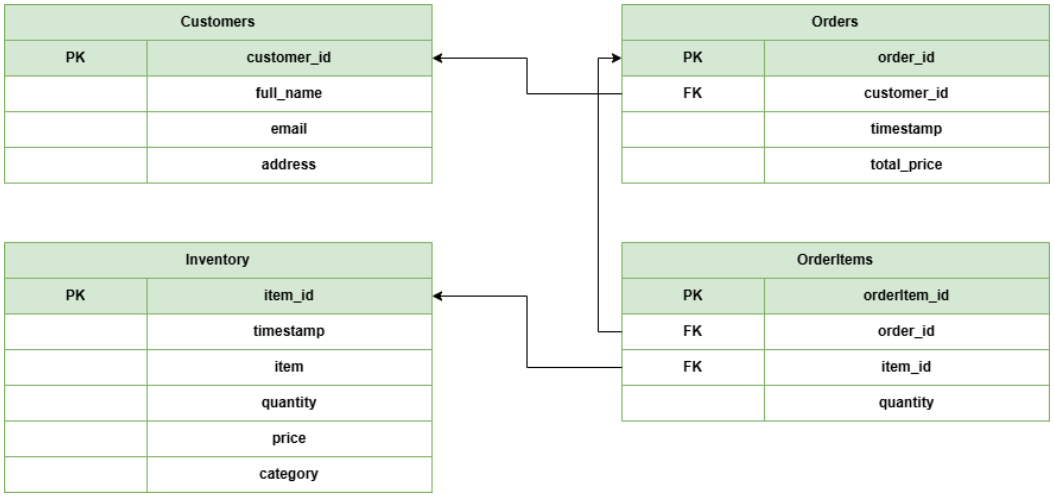

Figure1: Illustration of the considerations and up front planning of the SQL database schema design before implementation.

Figure 2-5 is an example of how the table data could look like in SQL.

	Customer_Id	FullName	Email	Address	Mobile	Birthday
1	1	someone1	someone1@example.com	address1	NULL	NULL
2	2	someone2	someone2@example.com	address2	NULL	NULL
3	3	someone3	someone3@example.com	address3	NULL	NULL
4	4	someone4	someone4@example.com	address4	1234	ddmmyyyy
5	5	someone5	someone5@example.com	address5	1234	ddmmyyyy

Figure2: Example of the customer table, where row 4-5 are inserted after the 1year migration.

	Item_Id	Timestamp	Item	Quantity	Price	Category
1	1	2024-12-30 23:33:35.300	item1	100	100.00	category1
2	2	2024-12-30 23:33:35.300	item2	90	110.00	category1
3	3	2024-12-30 23:33:35.300	item3	40	180.00	category2
4	4	2024-12-30 23:33:35.300	item4	30	190.00	category2
5	5	2024-12-30 23:33:35.313	item1	99	100.00	category1
6	6	2024-12-30 23:33:35.313	item2	88	110.00	category1
7	7	2024-12-30 23:33:35.313	item3	39	180.00	category2
8	8	2024-12-30 23:33:35.313	item4	29	190.00	category2
9	9	2024-12-30 23:33:35.493	item1	98	100.00	category1
10	10	2024-12-30 23:33:35.493	item2	86	110.00	category1

Figure3: Example of the inventory table. Row 1-4 are the initial inventory. Row 5-6 show the inventory update of the purchased items in order1. Similarly, rows 7-8 and 9-10 show the inventory update of the purchased items in order2 and order3, respectively. Note order3 is identical to order1 but where a discount code has been added from the 1year migration.

	Order_Id	Customer_Id	Timestamp	Total_price
1	1	1	2024-12-30 23:33:35.313	320.00
2	2	2	2024-12-30 23:33:35.313	370.00
3	3	1	2024-12-30 23:33:35.493	160.00

Figure4: Example of orders, where timestamp is the timestamp of the purchased order and total_price is the total price of the entire order.

	OrderItem_id	Order_id	Item_id	Quantity	DiscountCode
1	1	1	1	1	NULL
2	2	1	2	2	NULL
3	3	2	3	1	NULL
4	4	2	4	1	NULL
5	5	3	1	1	black_friday50
6	6	3	2	2	black_friday50

Figure5: Example of orderItems. Row 5-6 corresponds to order3 after the 1year migration, where a discount code has been added to the order data.

Figure 6-7 illustrates how the complete order information can be shown by SQL query. This corresponds to the information that would be stored in order data in MongoDB where we embed all relevant information. The SQL scripts that query this combined order data are located here:

- SQL\ Query_full_order_details.sql
- SQL\Query_full_order_details_post_migration.sql

where the SQL folder is located in the same folder as this document.

	Order_Id	customer_id	full_name	email	address	item_id	item	price	category	amount	Total_price	Timestamp
1	1	1	someone1	someone1@example.com	address1	1	item1	100.00	category1	1	320.00	2024-12-30 23:33:35.313
2	1	1	someone1	someone1@example.com	address1	2	item2	110.00	category1	2	320.00	2024-12-30 23:33:35.313
3	2	2	someone2	someone2@example.com	address2	3	item3	180.00	category2	1	370.00	2024-12-30 23:33:35.313
4	2	2	someone2	someone2@example.com	address2	4	item4	190.00	category2	1	370.00	2024-12-30 23:33:35.313

Figure6: Entire order data information of order1 and order2, before the 1year migration.

	Order_Id	customer_id	full_name	email	address	item_id	item	price	category	amount	discount_code	Total_price	Timestamp
1	3	1	someone1	someone1@example.com	address1	1	item1	100.00	category1	1	black_friday50	160.00	2024-12-30 23:33:35.493
2	3	1	someone1	someone1@example.com	address1	2	item2	110.00	category1	2	black_friday50	160.00	2024-12-30 23:33:35.493

Figure7: Entire order data information of order3, after the 1year migration.

Figure 8-9 shows the minimalistic migrations implemented in SQL. Note, tests, rollback plans and tests of rollback plans etc. have not been implemented.

```sql
-- begin transaction (ACID compliant)
begin try
    begin transaction;

    -- if column Mobile does not exists in Customers table
    if not exists (select * from INFORMATION_SCHEMA.COLUMNS where TABLE_NAME = 'Customers' and COLUMN_NAME = 'Mobile' and TABLE_SCHEMA = 'dbo')
    begin
        alter table Customers
        add Mobile int,
            Birthday nvarchar(100);
    end

    commit transaction;
end try
begin catch
    -- rollback changes in case of an error
    rollback transaction;

    -- print errors
end catch
```

Figure8: Minimalistic migration of Customers table after 1year using transactions, i.e. fields Mobile and Birthday are added. This is done in file SQL\011_migration_customers_after1year.sql

```sql
-- begin transaction (ACID compliant)
begin try
    begin transaction;

    if not exists (select * from INFORMATION_SCHEMA.COLUMNS where TABLE_NAME = 'OrderItems' and COLUMN_NAME = 'DiscountCode' and TABLE_SCHEMA = 'dbo')
    begin
        alter table OrderItems
        add DiscountCode nvarchar(100);
    end

    commit transaction;
end try
begin catch
    -- rollback changes in case of an error
    rollback transaction;

    -- print errors
end catch
```

Figure9: Minimalistic migration of OrderItems table after 1year using transactions, i.e. field discountCode is added. This is done in file SQL\012_migration_orderItems_after1year.sql

For comparison to figures 8-9, figure 10 shows how easy and flexible a migration in MongoDB can be. Basically, all that needs to be done is adding the additional fields. Despite being optional, one should of course impose some structure, e.g. as in the case for logging data, as frequent changes or updates to the fields without any structure can make querying very difficult later on.

```python
import uuid

from CreateDB import db

# Opret en collection for customers
customers_collection = db["customers"]

# Indsæt data/dokumenter
customers_data = [
    {"customer_id": str(uuid.uuid4()), "full_name": "someone1", "email": "someone1@example.com", "address": "address1"},
    {"customer_id": str(uuid.uuid4()), "full_name": "someone2", "email": "someone2@example.com", "address": "address2"},
    {"customer_id": str(uuid.uuid4()), "full_name": "someone3", "email": "someone3@example.com", "address": "address3"},
]

# Indsæt data i collection
customers_collection.insert_many(customers_data)


# Migration efter 1år
customers_data2 = [
    {"customer_id": str(uuid.uuid4()), "full_name": "someone4", "email": "someone4@example.com", "address": "address4", "mobile": 1234, "birthday": "ddmmyyyy"},
    {"customer_id": str(uuid.uuid4()), "full_name": "someone5", "email": "someone5@example.com", "address": "address5", "mobile": 1234, "birthday": "ddmmyyyy"},
]

# Indsæt data i collection
customers_collection.insert_many(customers_data2)
```

Figure10: Example of how to construct the customers schema before and after the 1year migration. Note, due to the schema-less and flexible structure we can just add the additional key-value pairs mobile and birthday.

Figure 11 shows an example of how order data with embedded customer information could look like in MongoDB.

```
Documents in collection 'orders':
{'_id': ObjectId('677325801d20f671f6f07105'),
 'customer': {'address': 'address1',
              'customer_id': '1d6d6a0f-e64a-4868-965c-6d58d12a086e',
              'email': 'someone1@example.com',
              'full_name': 'someone1'},
 'items': [{'amount': 1,
            'category': 'category1',
            'item': 'item1',
            'item_id': '6b5183ef-dc3c-42cd-94c3-5be64dd836e4',
            'price': 100},
           {'amount': 2,
            'category': 'category1',
            'item': 'item2',
            'item_id': 'e49e91ad-972c-48c2-8dc3-d73de6ff8f50',
            'price': 110}],
 'order_id': '73e098e7-976d-4b58-9196-fe0448509e1d',
 'timestamp': datetime.datetime(2024, 12, 30, 22, 58, 8, 80000),
 'total_price': 320}
```

Figure11: Example of how order1 and order2 could look with embedded customer information.