Design consideration:

Our sharded cluster components:

Two shards (replica sets)

Shard1 : ports : 27117, 27127, 27137

Shard2: ports: 27217, 27227, 27237

Congif replica set

Ports : 30117, 30127, 30137

The default port is reserved to the query router, and mongos instance has been used for this job

The replica sets for the shards have the following configuration

members:

{\_id:0,host:'localhost:27217',votes:1},

{\_id:1,host:'localhost:27227',slaveDelay:20,priority:0,votes:1},

{\_id:2,host:'localhost:27237',votes:1}

Every instance has the right to vote in an election scenario, one server will hold a backup delayed version of the data to save us from potential deadly queries like dropping collections.

The delayed server will be hidden, so that the client does not read outdated data.

Sharding kye:

sh.shardCollection('assignment.book',{ISBN:1,title:1})

for the book collection a sorted key has been used to ensure the uniqueness of the key as well as hight cardinality. Thus, the chunks will be distributed evenly accros the shards, the balancing process is enabled by default for all our shards .

sh.shardCollection('assignment.order',{\_id:'hased'}) hashed shard key, on \_id

for the order collection a hashed key has been used since the id is a monotonical value and cannot be used as a sorted key.

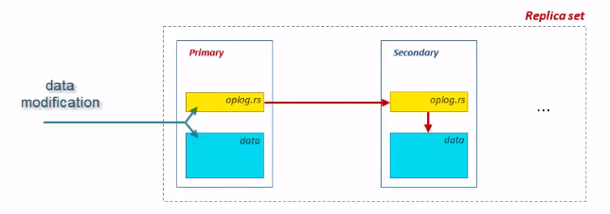
1. replica sets

Each replica set consists of three instances running on the same machine but utilising different ports. Odd number of servers can achieve an absolute majority in the election scenario, and we decided to use the minimum number of voting members. The backup server which holds the outdated data cannot be a primary server. To ensure that the delayed server would not be a primary server, we have assigned a 0 priority for it. Thus, one of the remaining two servers can be a primary.

Writing and reading policy:

Any data modification should happen on the primary instance, which will be transferred to the secondary’s afterwards. The delayed server will read the operations from the oplog.rs collection after 20 seconds. The default size has been used for the oplog collection. Since we do not have a big set of data in our case, the default 5% of free disk space is enough to use for the oplog collection.

The reading operations from the oplog collection might take some time and that is why the data are eventually consistent. Since the oplog is a capped collection, it means that it can hold a limited number of operations. In a network partition scenario: if one server could not read from the oplog collection for a certain amount of time, and in the meanwhile many operations have been added to the oplog collection, the following will happen: the oplog will overwrite some of the previous operations, if it runs out of space, which will cause data inconsistency in the replica set



Benefits from Replica Set

Automatic failover

The ability to automatically handle breakdowns of one or more of the replica members.

Scalability

The primary is the only writable instance, and the client can read from secondaries, but not including the backup server. Thus, we can perform many more reads against the replicas rather than attacking a single server with all our requests to write and read.

NoSQL VS Relational database

Locking and Scalability

In the relational model, we ensure consistency and atomicity by allowing one write access at a time. This technique can be applied by locking rows and tables. Protecting referential integrity across multiple tables and rows increases the time the lock must be in effect. Thus, less writing and less updating per second, which leads to a higher latency of transactions. In the case of replicating the data across many servers, it can make things worse because the database engine will enforce extending the locks across the network, which leads to higher latency.

The locking in Mongo is much simpler, in the case of schema less collections, because mongo has a single document writing scope, and updating documents occur one at a time. Furthermore, no need to extend locking on multiple collections since there are no relationships to enforce and no schema to protect. In our assignment where mongo has a schema, we enforced data consistency by using two phase commit technique. The important part is that the engine will never extend locking on multiple collections. This has been said, the latency time will be better. When it comes to scalability, mongo is a better choice because it does not let you lock across servers. A replica set in mongo consists of one primary that accepts all rights, and several secondaries which will be replicated versions of the primary. The main difference here is that the locks do not extend from the primary to the secondaries.

However, in mongo we can still decide the consistency model we want. For instance, we can make an update command to the primary and wait until the document has been replicated to all servers. Or we can choose to only wait for the primary to persist the document. We can also choose that most of the servers have replicated the document before the write was acknowledged. Moreover, we can even choose to hand over the document to primary and not care whether it was persisted or not.

Transactions

In a relational database managing transactions is ideal for situations where databases, regardless of where they sit, need atomicity, consistency, isolation, and durability.

In mongo, however, the developer needs to deal with this scenario manually, which is not a good thing for our case in this assignment. Since we are dealing with orders and updating documents in more than one collection, managing data consistency was quite challenging to achieve. The transaction framework introduced in mongo 3.2 is not easy to use and it requires complex configuration.

If it is up to me, I will choose relational database over mongo for this particular exercise, mainly because we need to modify the data all the time and manage transactions across multiple collections.

Mongo is extremely fast for reading and writing, and it is the best choice when it come to the vertical scaling. If we want to scale horizontally, like we did with the shards in this assignment, SQL is much harder to achieve. One thing to notice though, when we apply a strict schema to mongo, we will lose a lot of its flexibility. Moreover, we had to force some normalization in a way that mongo is not intended to deal with in the first place. Mongo will be a very good choice when we are dealing with data that can grow fast and we do not know the shape of the data in advance. However, in this assignment SQL will be a better choice since data organization can be achieved easily.

Data manipulation

Mongo has a Low level query language: while NoSQL databases have extended NoSQL to allow for joins, NoSQL does not have the versatility, or the interoperability of SQL.

Using the aggregation pipeline, however, was much easier than SQL queries regarding data manipulation, and the same applies for the recursive queries. graphLookup was a straightforward process, while the recursive with clause was as complicated as it can be.

Schema design

Customer and order relationship is that one customer has many orders:

The possibilities of mapping are:

1: reference id

* 1. : customer has an array of references (IDs) from the order collection:

-Reading is slow, because we need to make a lookup from two collections

-updates must happen through a two phase commit,

- data integrity has to be taken in consideration (adding an order ID which does not exist in the order collection might cause inconsistency)

-the size is not an issue, because the order reference can hold 1.4 million references of type object id (the document in total cannot succeed 14 MB)

(good option after all)

* No repeated order numbers, since each order belongs to one customer
  1. : order contains a customer ID:

-Reading is slow, because we need to make a lookup from two collections

-updating has to happen through a two phase commit.

- data integrity the same as in point 1.1

- the size is not an issue as well because we have only one customer

- redundant data: so many duplicated references

-there will be some repeated customer IDs in the collection, so the redundancy level is higher than the first option but still acceptable

The final decision depends on how often we will query towards the customer collection.

If the use case is : find all customers who have no orders. Then it will be easier if we have a reference in the customer collection, and search for null values there rather than if we have a customer reference in the order collection . In this case, we need to find all customer IDs in the order collection and compare them with all customer IDs from the customer collection, then the difference will be the customers who have no orders.

2: embedded document

2.1 Order collection embeds the customer object as a subdocument

-So many redundant sub-documents (bad choice)

-Atomic update (good, but at the expense of the redundant data)

2.2Customer collection embeds and array of order objects

The size of the document will be an issue in this case (bad choice)

final decision

Customer document will contain an array of order numbers, and an embedded document about address details

Order and book :

Options

References or embedded

1: reference

Book contains an array of references about orders: (bad idea : size matters)

The size matters here because a book can be involved in many orders (more than a million copies can be sold for one particular book)

Order contains an array of book references: (good idea)

No size issues.

No atomic update, but we can come around this by implementing a two phase commit

Data integrity: we need to make sure that we map existing IDs when we add the reference

2: embedded document

Order contains an embedded document about book:(bad idea)

(redundancy issue)

Book contains an embedded order document (bad idea: redundancy)

Final decision

Order contains an array of references from the book collection

The order document will have a price field for historical purposes