جامعة طرابلس

كلية تقنية المعلومات

قواعد البيانات النقالة والغير متجانسة

Heterogeneous and Mobile Databases

ITMC322

أستاذ المادة / محمد أوهيبة

المحاضرة السابعة

Review of topics

- *Introducing the write operations*
  - *Inserts*
  - *Updates*
- *Write concerns*
- *Bulk writing documents*
  - *Indexing documents*
    - *Indexing a single field*
    - *Indexing more than one field*

Introducing the write operations

- In MongoDB, we have three kinds of write operations: insert, update, and remove.
- these operations, MongoDB provides three interfaces: db.document.insert, db.document.update, and db.document.remove.
- The write operations in MongoDB are targeted to a specific collection and are **atomic** on the level of a single document.
- MongoDB uses a journaling mechanism to write operations, and this mechanism uses a journal to write the change operation before we write it in the data files. This is very useful, especially when we have a dirty shutdown.
- MongoDB will use the journal files to recover the database state to a consistent state when the mongod process is restarted.

Inserts

The insert interface is one of the possible ways of creating a new document in MongoDB. The insert interface has the following syntax:

```
db.collection.insert(
<document or array of documents>,
{
writeConcern: <document>,
ordered: <boolean>
}
)
```

Here:

- document or array of documents is either a document or an array with one or many documents that should be created in the targeted collection.

- writeConcern is a document expressing the write concern.

- ordered should be a Boolean value, which if true will carry out an ordered process on the documents of the array, and if there is an error in a document, MongoDB will stop processing it. Otherwise, if the value is false, it will carry out an unordered process and it will not stop if an error occurs. By default, the value is true.

Inserts (Cont.)

- In the following example, we can see how an insert operation can be used:

```
db.customers.insert({
username: "customer1",
email: "customer1@customer.com",
password: hex_md5("customer1paswd")
})
```

- As we did not specify a value for the _id field, it will be automatically generated with a unique ObjectId value. The document created by this insert operation is:

```
{
"_id" : ObjectId("5487ada1db4ff374fd6ae6f5"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1"
}
```

- in the first example of this section, the insert interface is not the only way to create new documents in MongoDB. By using the upsert option on updates, we could also create new documents.

Updates

- The update interface is used to modify previous existing documents in MongoDB.
- or even to create new ones.
- An update operation will modify only one document at a time. If the criterion matches more than one document, then it is necessary to pass a document with a multi parameter with the true value to the update interface.
- If the criteria matches no document and the upsert parameter is true, a new document will be created, or else it will update the matching document.
- The update interface is represented as:

```
db.collection.update(
    <query>,
    <update>,
    {
        upsert: <boolean>,
        multi: <boolean>,
        writeConcern: <document>
    }
)
```

Updates (Cont.)

- Here:
- **query** is the criteria

- **update** is the document containing the modification to be applied

- **upsert** is a Boolean value that, if true, creates a new document if the criteria does not match any document in the collection

- **multi** is a Boolean value that, if true, updates every document that meets the criteria
- **writeConcern** is a document expressing the write concern

- Using the document created in the previous session, a sample update would be:

```
db.customers.update(
{username: "customer1"},
{$set: {email: "customer1@customer1.com"}}
)
```

- The modified document is:

```
{
"_id" : ObjectId("5487ada1db4ff374fd6ae6f5"),
"username" : "customer1",
"email" : "customer1@customer1.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1"
}
```

Updates (Cont..)

- The $set operator allows us to update only the email field of the matched documents. Otherwise, you may have this update:

```
db.customers.update(
{username: "customer1"},
{email: "customer1@customer1.com"}
)
```

- In this case, the modified document would be:

```
{
"_id" : ObjectId("5487ada1db4ff374fd6ae6f5"),
"email" : "customer1@customer1.com"
}
```

- That is, without the $set operator, we modify the old document with the one passed as a parameter on the update.

- we also have other important update operators:
- $inc increments the value of a field with the specified value:

```
db.customers.update(
{username: "johnclay"},
{$inc: {"details.age": 1}}
```

- This update will increment the field details.age by 1 in the matched documents.

Updates (Cont...)

- $rename will rename the specified field:

```
db.customers.update(
{email: "customer1@customer1.com"},
{$rename: {username: "login"}}
)
```

- This update will rename the field username to login in the matched documents.

- $unset will remove the field from the matched document:

```
db.customers.update(
{email: "customer1@customer1.com"},
{$unset: {login: ""}}
)
```

- This update will remove the login field from the matched documents.

- As the write operations are atomic at the level of a single document, we can afford to be careless with the use of the preceding operators. All of them can be safely used.

Write concerns

- Many of the discussions surrounding non-relational databases are related to the **ACID** concept. (**a**tomicity, **c**onsistency, **i**solation, and **d**urability).

- we should understand by now why we really have to take this matter into consideration, and how these simple four letters are essential in the non-relational world.
- we will discuss the letter **D**, which means **durability**, in MongoDB.

- **Durability** in database systems is a property that tells us whether a write operation was successful, whether the transaction was committed, and whether the data was written on non-volatile memory in a durable medium, such as a hard disk.

- Unlike relational database systems, the response to a write operation in NoSQL databases is determined by the client.

- we have the possibility to make a choice on our data modeling, addressing the specific needs of a client.

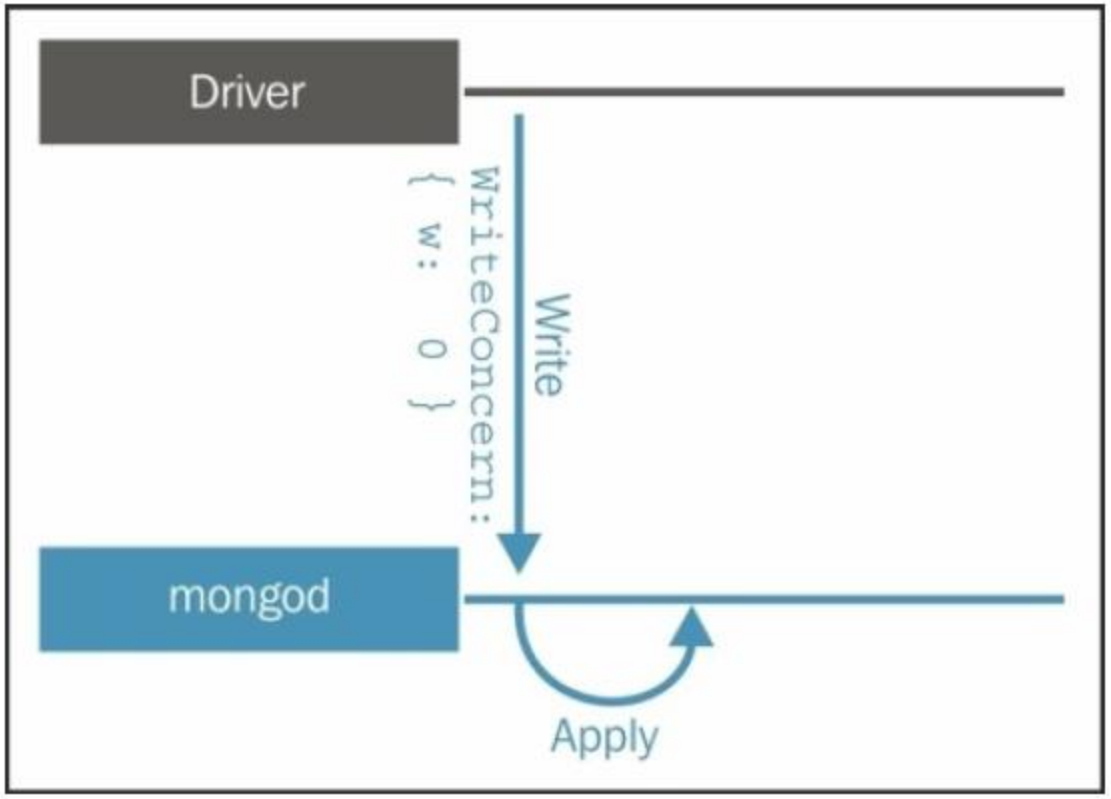
Write concerns (Cont.)

- In MongoDB, the response of a successful write operation can have many levels of guarantee.
- This is what we call a write concern. The levels vary from weak to strong, and the client determines the strength of guarantee.
- It is possible for us to have, in the same collection, both a client that needs a strong write concern and another that needs a weak one.
- The write concern levels that MongoDB offers us are:
  - Unacknowledged
  - Acknowledged
  - Journaled
  - Replica acknowledged

Write concerns (Cont..)

- **Unacknowledged**
- As its name suggests, with an unacknowledged write concern, the client will not attempt to respond to a write operation. If this is possible, only network errors will be captured.

- The following diagram shows that drivers will not wait that MongoDB acknowledge the receipt of write operations:



- In the following example, we have an insert operation in the customers collection with an unacknowledged write concern:

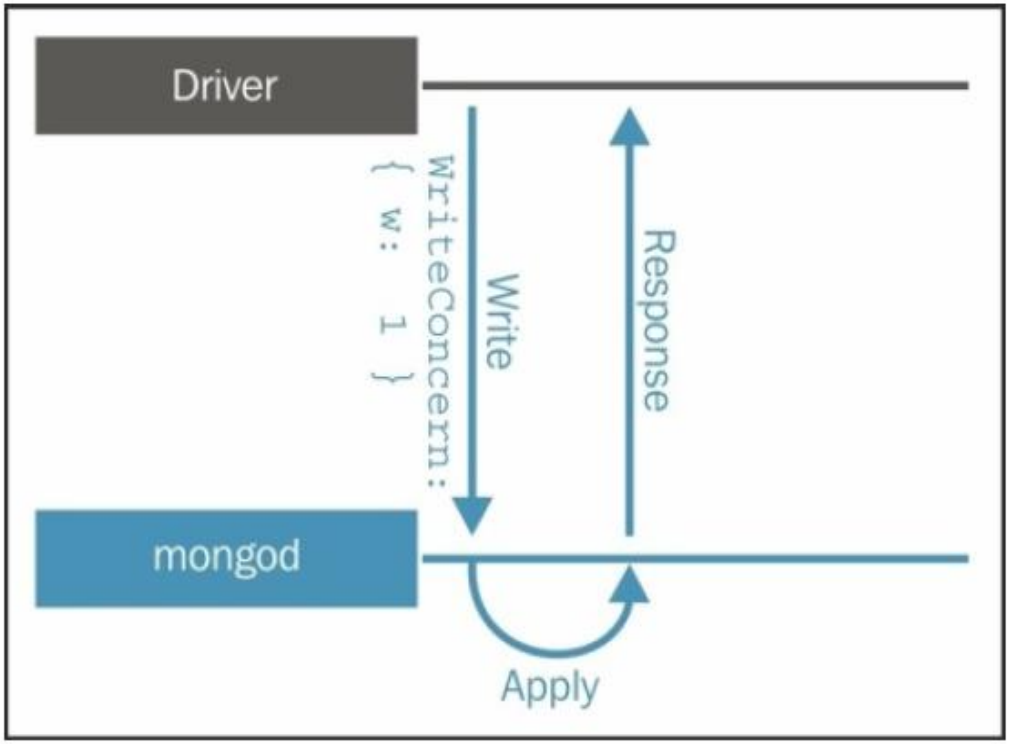
```
db.customers.insert(
{username: "customer1",
email:
"customer1@customer.com
", password:
hex_md5("customer1paswd
")},
{writeConcern: {w: 0}}
)
```

Write concerns (Cont...)

- **Acknowledged**
- With this write concern, the client will have an acknowledgement of the write operation, and see that it was written on the in-memory view of MongoDB.

- In this mode, the client can catch, among other things, network errors and duplicate keys. Since the 2.6 version of MongoDB, this is the default write concern.

- As you saw earlier, we can't guarantee that a write on the in-memory view of MongoDB will be persisted on the disk. In the event of a failure of MongoDB, the data in the in memory view will be lo
- The following diagram shows that dr write operations and applied the char



- In the following example, we have an insert operation in the customers collection with an acknowledged write concern:

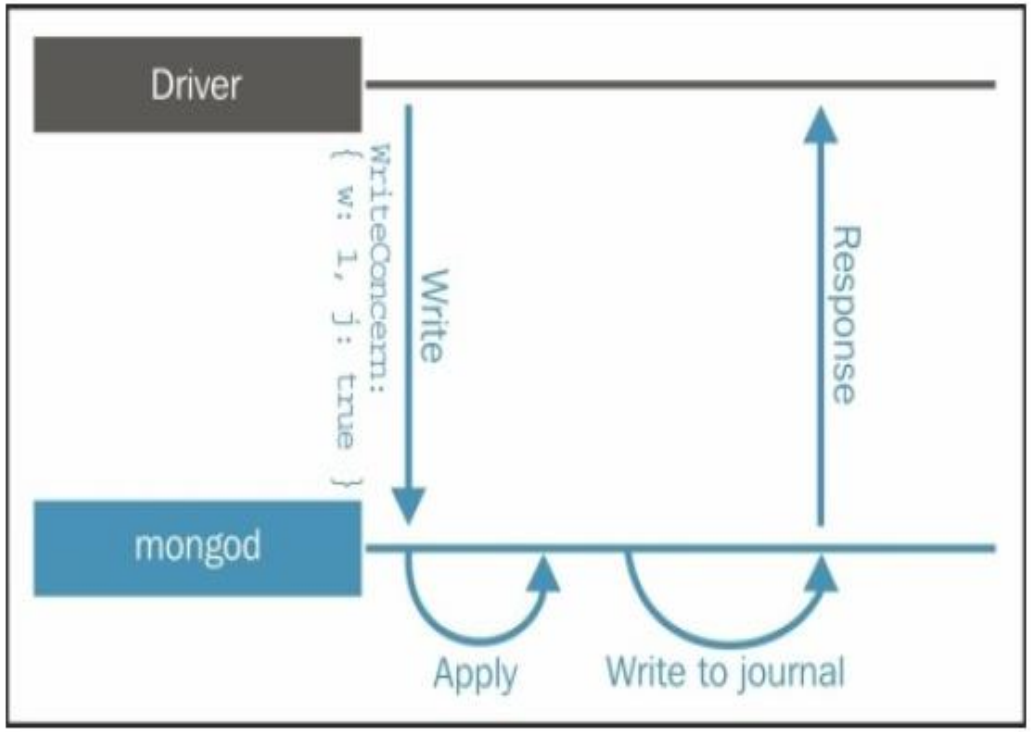
```
db.customers.insert(
{username: "customer1",
email:
"customer1@customer.com",
password:
hex_md5("customer1paswd")},
{writeConcert: {w: 1}}
)
```

Write concerns (Cont....)

- **Journaled**
- With a journaled write concern, the client will receive confirmation that the write operation was committed in the journal. Thus, the client will have a guarantee that the data will be persisted on the disk, even if something happens to MongoDB.

- To reduce the latency when we use a journaled write concern, MongoDB will reduce the frequency in which it commits operations to the journal from the default value of 100 milliseconds to 30 milliseconds.

- The following diagram shows that drivers will wait MongoDB acknowledge the receipt of write operations only after cor



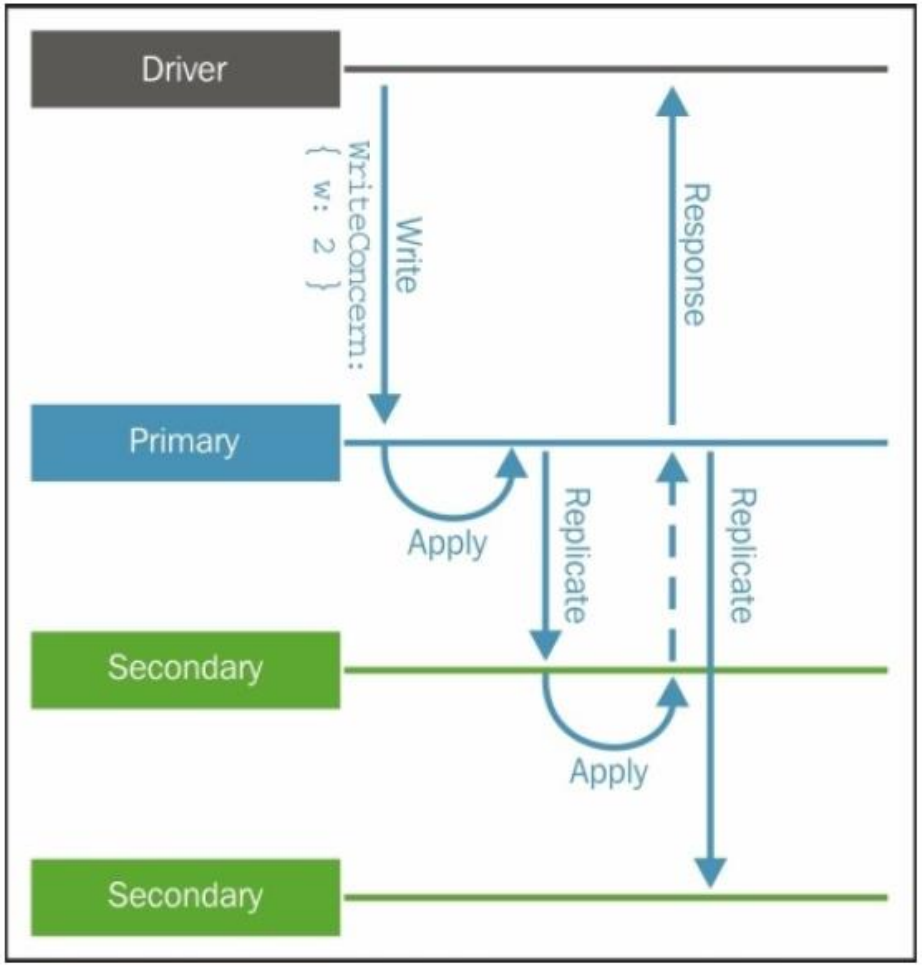
- In the following example, we have an insert in the customers collection with a journaled write concern:

```
db.customers.insert(
{username: "customer1", email:
"customer1@customer.com",
password:
hex_md5("customer1paswd")},
{writeConcern: {w: 1, j: true}}
)
```

Write concerns (Cont…..)

- **Replica acknowledged**
- When we are working with replica sets, it is important to be sure that a write operation was successful not only in the primary node, but also that it was propagated to members of the replica set. For this purpose, we use a replica acknowledged write concern.

- By changing the default write concern to r the number of members of the replica set f confirmation.
- The following diagram shows that drivers receipt of write operations on a specified n



In the following example, we will wait until the write operation propagates to the primary and at least two secondary nodes:

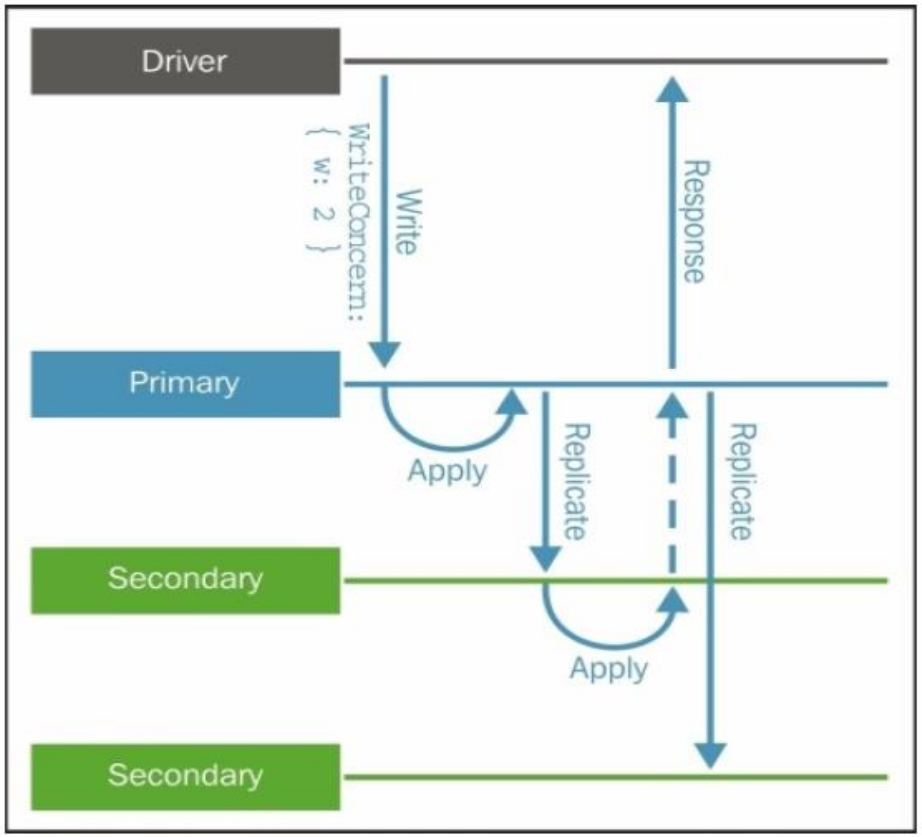
```
db.customers.insert(
{username: "customer1", email:
"customer1@customer.com",
password:
hex_md5("customer1paswd")},
{writeConcern: {w: 3}}
)
```

Write concerns (Cont.....)

- We should include a timeout property in milliseconds to avoid that a write operation remains blocked in a case of a node failure.

- In the following example, we will wait until the write operation propagates to the primary and at least two secondary nodes, with a timeout of three seconds. If one of the two secondary nodes from which we are expecting a response fails, then the method times out after three seconds:

```
db.customers.insert(
{username: "customer1", email:
"customer1@customer.com", password
hex_md5("customer1paswd")},
{writeConcern: {w: 3, wtimeout:
3000}}
)
```



Bulk writing documents

- Sometimes it is quite useful to insert, update, or delete more than one record of your collection.
- MongoDB provides us with the capability to perform bulk write operations.

- bulk operation works in a single collection, and can be either ordered or unordered.

- As with the insert method, the behavior of an ordered bulk operation is to process records serially, and if an error occurs, MongoDB will return without processing any of the remaining operations.

- The behavior of an unordered operation is to process in parallel, so if an error occurs, MongoDB will still process the remaining operations.

- We also can determine the level of acknowledgement required for bulk write operations.
- we can make a bulk insert only by passing an array of documents on the insert

Bulk writing documents (Cont.)

- method. In the following example, we make a bulk insert using the insert method:

```
db.customers.insert(
[{
username: "
customer3", email: "
customer3@customer.com", password:
hex_md5("customer3paswd")},
{username: "customer2", email: "customer2@customer.com", password:
hex_md5("customer2paswd")},
{username: "customer1", email: "customer1@customer.com", password:
hex_md5("customer1paswd")}
])
```

In the following example, we make an unordered bulk insert using the new bulk methods:

```
var bulk = db.customers.initializeUnorderedBulkOp();
bulk.insert({username: "customer1", email: "customer1@customer.com",
password: hex_md5("customer1paswd")});
bulk.insert({username: "customer2", email: "customer2@customer.com",
password: hex_md5("customer2paswd")});
bulk.insert({username: "customer3", email: "customer3@customer.com",
password: hex_md5("customer3paswd")});
bulk.execute({w: "majority", wtimeout: 3000});
```

- MongoDB has a limit of executing a maximum of 1,000 bulk operations at a time. So, if this limit is exceeded, MongoDB will divide the operations into groups of a maximum of 1,000 bulk operations.

Reference Book

- **`MongoDB Data Modeling`** **Wilson da Rocha França**

The End

Thanks for listening ..

Any questions ?