VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

MongoDB
Distributed Write and Read

Lecturer : Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Plan for Distributed Write and Read

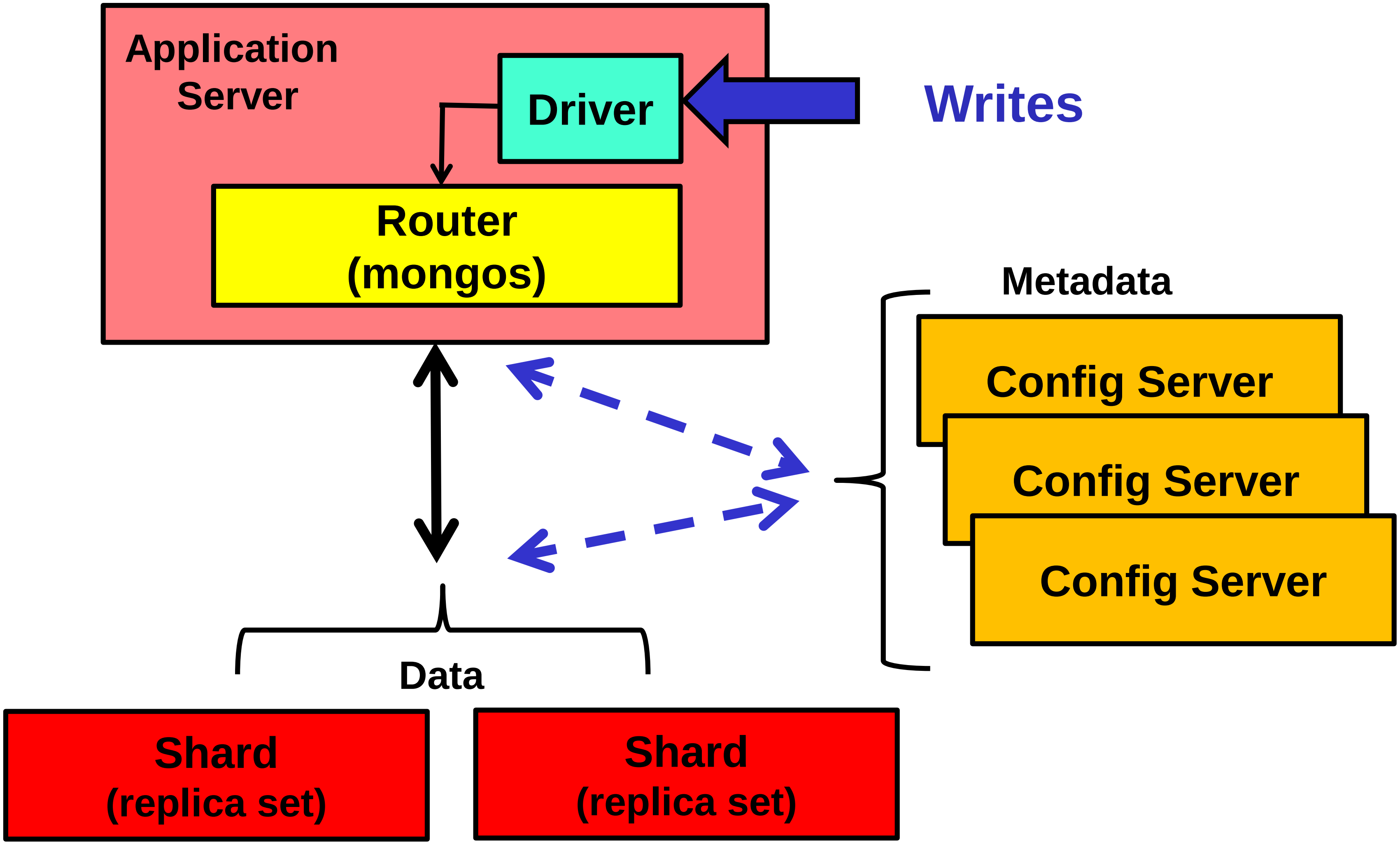
- Distributed Write
  - Write on Sharded Cluster
  - Write on Replica Sets
- Write Concern
  - `Bulk()` Method
- Distributed Queries
- MongoDB and Transaction Processing

  - ***Reedings:***
    - *Have a look at Readings on the Home Page*

Write Operations on Sharded Clusters

- For sharded collections in a sharded cluster, the `mongos` directs write operations from applications to shards that are responsible for the portion of the data set using the sharding key value
- The `mongos` gets needed metadata information from the config database residing on config servers

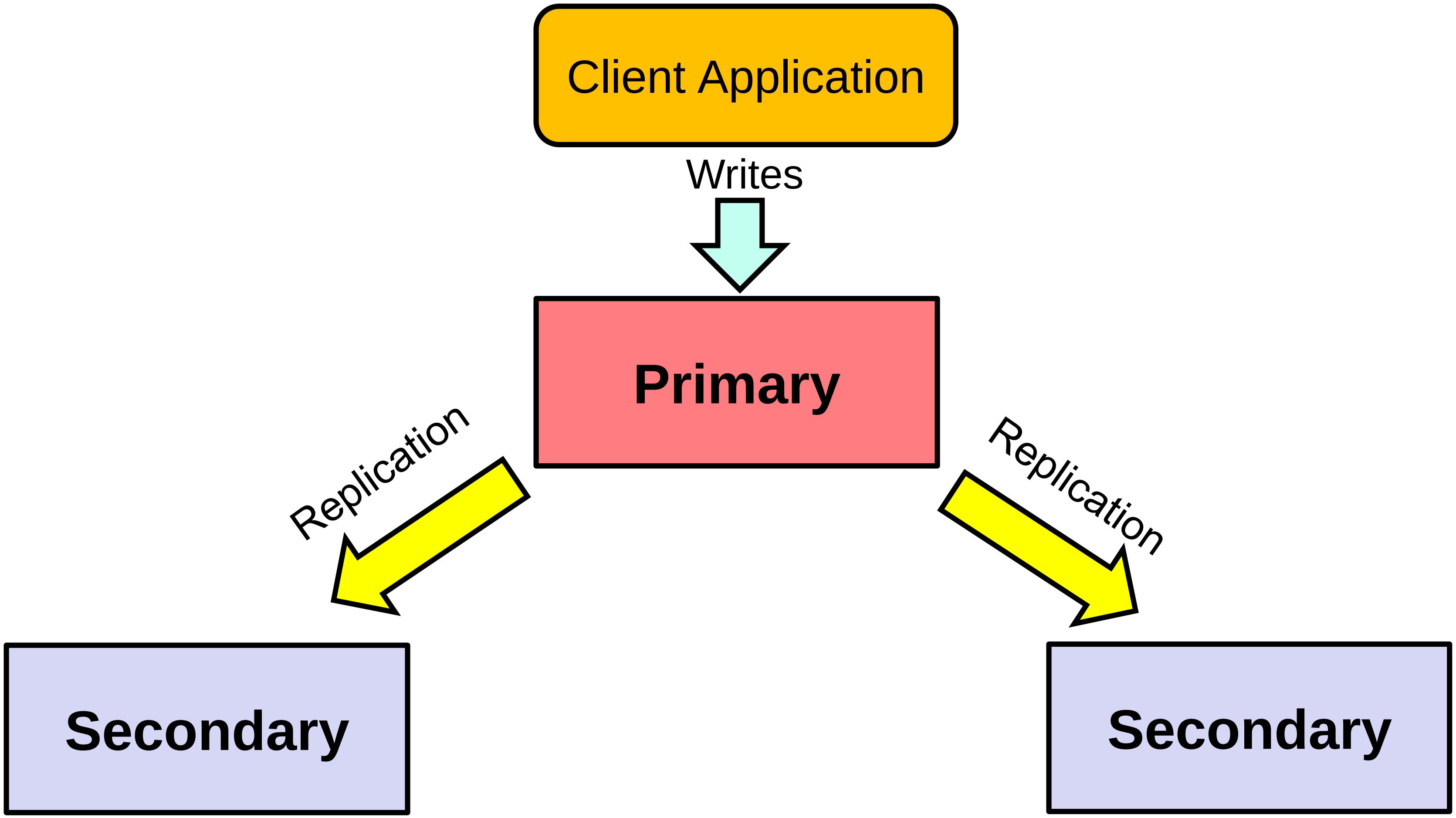
Sharded Cluster



Write Operations on Replica Sets

- In replica sets, all write operations go to the set's primary
- The primary applies the write operations and then records the operations on its operation log (oplog)
  - Oplog is a reproducible sequence of operations to the data set
- Secondary members of the set continuously replicate the oplog by applying operations to themselves in an asynchronous process

Replica Set Operations



Write Concern (1)

- *Write concern* describes the guarantee that MongoDB provides when reporting on the success of a write operation
- The strength of the write concerns determine the level of guarantee
- When inserts, updates and deletes have a *weak* write concern, write operations return quickly
- In some failure cases, write operations issued with weak write concerns may not persist
- With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations

Write Concern (2)

- MongoDB (**version 2.6**) provides different levels of write concern:
  - Unacknowledged (lowest level),
  - Acknowledged (default),
  - Journaled, and
  - Replica Acknowledged (highest level)
- Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment
- For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment

Insert Multiple Documents with `Bulk()`

1. Initialize a `Bulk()` operator for the collection

```
var bulk =
  db.myclasses.initalizeUnorderedBulkOp();
```

2. Add a number of insert operations to the `bulk` object using `bulk.insert()` method

```
bulk.insert(doc1);
...
bulk.insert(docn);
```

3. Execute the `execute()` method on the `bulk` object

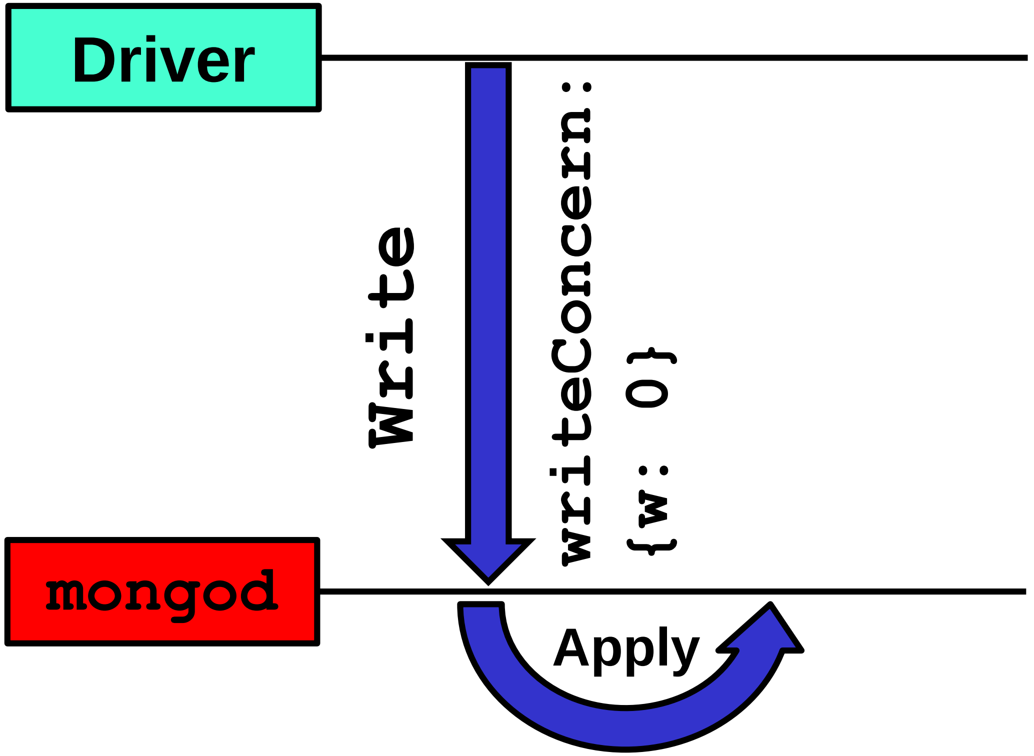
```
  bulk.execute({w: “j”});
```

   – The `execute()` method has an optional parameter **`w`** for specifying the write concern level

4. The method returns a `BulkWriteResult`

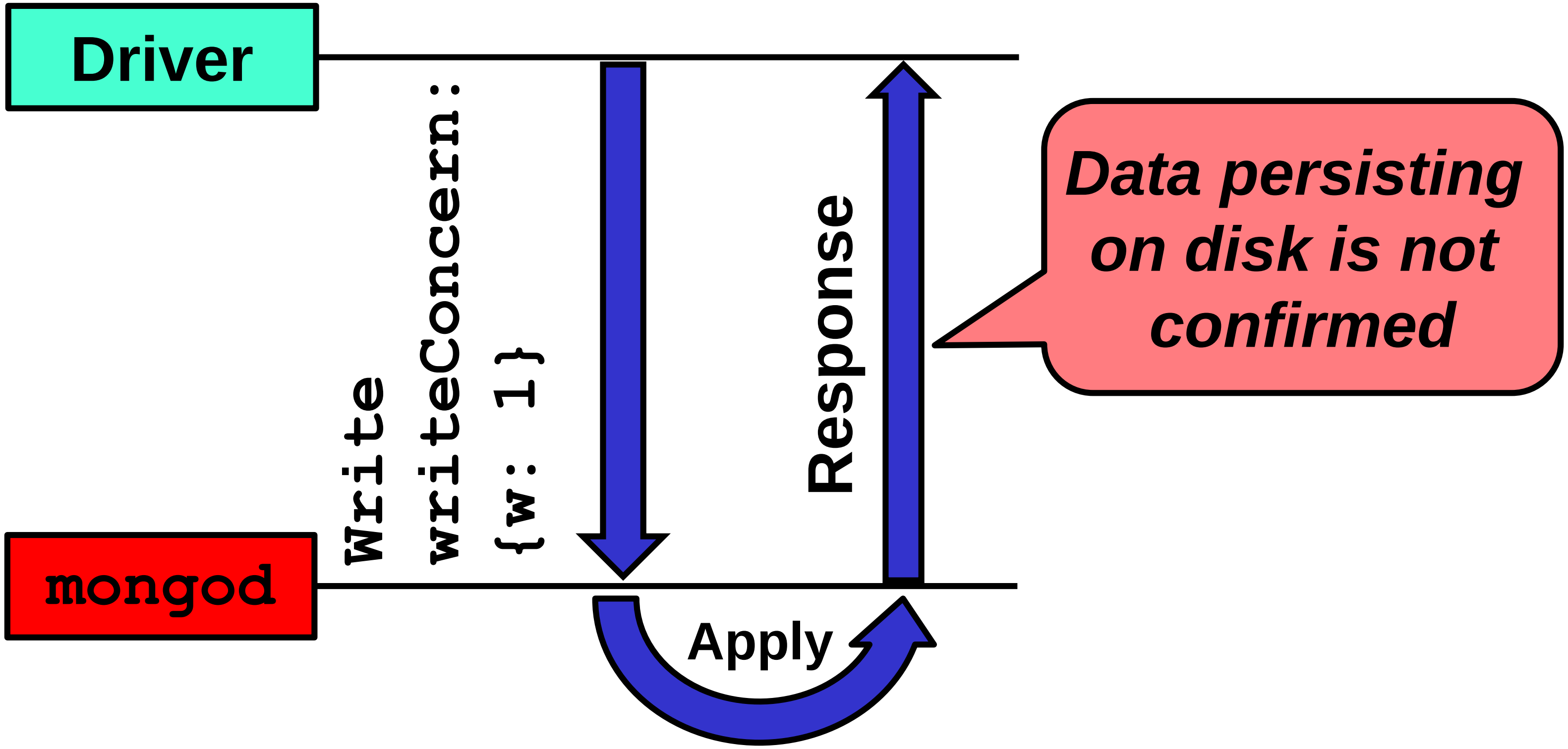
Write Concern: Unacknowledged

- If `{w: 0}`, MongoDB does not acknowledge the receipt of a write operation



Write Concern: Acknowledged

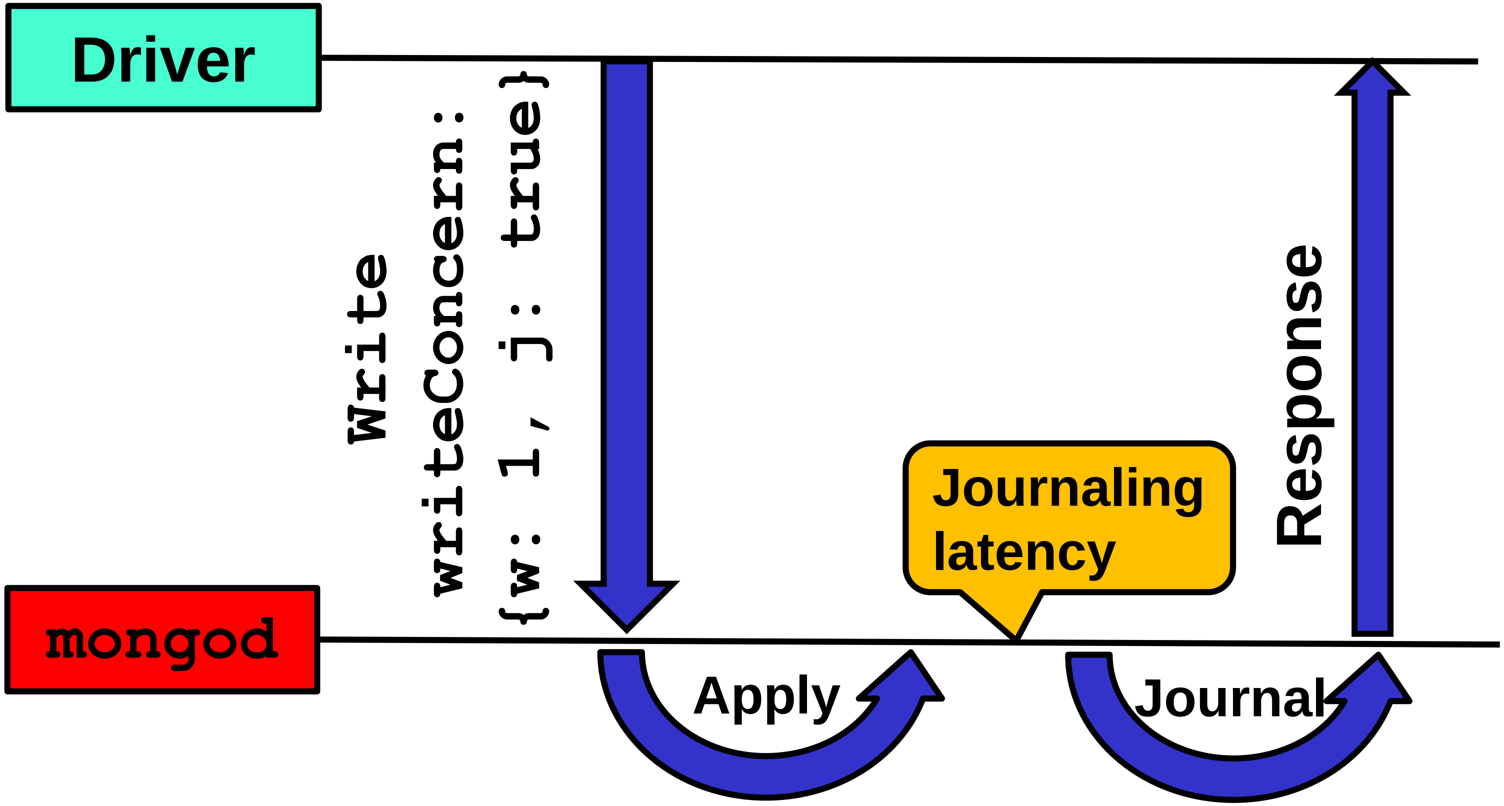
- If `{w: 1}`, MongoDB confirms that it applied a change to the in–memory data



Data persisting on disk is not confirmed

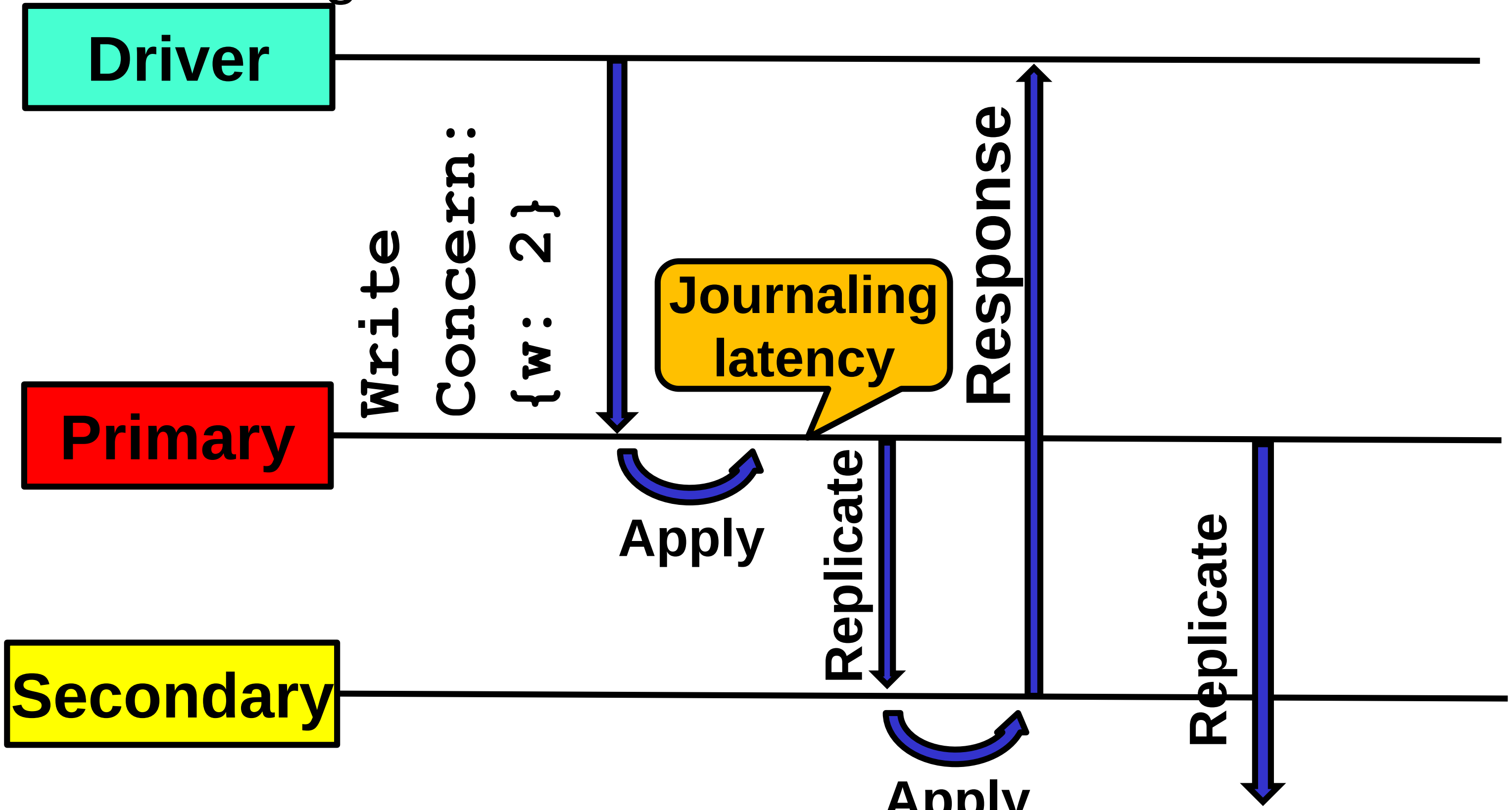
Write Concern: Journaled

- If `{w: 1, j}`, MongoDB confirms that it committed data on (master's) disk



Write Concern: Replica Acknowledged

- If `{w: 2}`, the first secondary to finish in memory application of primary's oplog operation, returns acknowledgment



Distributed Queries

- Applications issue operations to one of `mongos` instances of a sharded cluster
- Read operations are most efficient when a query includes the collection’s shard key
  - Otherwise the mongos must direct the query to all shards in the cluster (scatter gather query) and that might be inefficient
- By default, MongoDB always reads data from a replica set’s primary

Reading From a Secondary

- Reading from a secondary server is possible and justified if there is a need :
  - To balance the work load,
  - To allow reads during failover, but
  - Eventual consistency can be guaranteed, only
- To allow reading from a slave server, one of the following set-ups are needed:
  - Modifying the read preference mode in the driver, which results in a permanent change, or
  - Connecting to a slave server shell and issuing the following commands :

```
db.getMongo().setSlaveOk()
use <db_name>
db.collection.find()
```

Read Isolation

- MongoDB allows clients to read documents inserted or modified before committing modifications to disk, regardless of write concern level
  - MongoDB performs journaling frequently, but only after a defined time interval
- If the `mongod` terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the `mongod` restarts
  - This is a *read uncommitted* transaction anomaly.
- When `mongod` returns a successful *journaled write concern* (`"j"`), the data is fully committed to disk and will be available after `mongod` restarts

Atomicity

- A write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents *within* a single document
- When a single write operation modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic and other operations may interleave
  - There exists the `$isolated` operator that can *isolate* a single write operation
  - But it does **not** work on sharded clusters

Transaction Like Semantics

- Since a single document can contain multiple embedded documents, single-document atomicity is sufficient for many practical use cases
- For cases where a sequence of write operations must operate as if in a single transaction, a *two-phase commit* can be implemented in an application
- However, the two-phase commit can only offer transaction-*like* semantics
- Using two-phase commit ensures data consistency, but it is possible for applications to return intermediate data during the two-phase commit or rollback

Concurrency Control

- In relational databases, concurrency control allows multiple applications to run concurrently without causing data inconsistency or conflicts
- MongoDB does not offer such mechanisms
- Instead, there are techniques to avoid some sorts of inconsistencies:
  - Unique indexes used with certain methods like `findAndModify()` prevent duplicate insertions or updates
  - Also, there are certain programming patterns that can be applied to avoid concurrency control anomalies, like the lost update anomaly

Summary

- Routers direct client read and write operations to shards and their replica sets using meta data from config servers
- All writes go to the master server
- By default, all reads also go to the master server
- Write Concern is the guarantee that MongoDB provides when reporting on the success of a write operation
  - Week write concern: fast, but not very reliable
  - Strong write concern: slower, but more reliable
- By default, queries are of the type “read uncommitted”
- Queries based on the shard key value are the fastest
- Transaction like behavior is achievable to some extent