Resilient Algorithm Solution for MongoDB Applications

Ayush Jindal[1], Pavi Saraswat[1], Chandan Kapoor[1], and Punit Gupta[2](✉)

[1] Amity University, Noida, Uttar Pradesh 201313, India
ayushjindal15030@gmail.com, pavisaraswat@gmail.com, chandan.kapoor72@gmail.com
[2] JUIT, Wakhnaghat, Shimla, India
punitg07@gmail.com

Abstract. Algorithms and applications for large systems have usually assumed a fairly basic failure model: The computer is a reliable digital machine, with unchanging execution times and rare failures. If failure occurs, recovery can be handled by checkpoint-restart. With the push toward exascale computing, the concern of preserving the reliable, digital machine model will become too costly has become even greater, and we must focus towards Resilient programming to improve algorithms. In this Paper we discuss a Strategy for programming resilience in MongoDB.

Keywords: Resilient · MongoDB · Idempotent · Algorithm · Queries Non-idempotent

1 Introduction

Errors and random corruptions may affect the result computation of many modern machines. Resilient algorithms [1] and programming is a strategy that deals with network errors and outages and command errors. These are algorithms that are designed to perform in the presence of memory errors.

MongoDB is a leading NoSQL database, an open-source document and distributed database. MongoDB is a database that stores data in JSON-like documents. These documents can vary in structure. Through the MongoDB query language Related information is stored together for fast query access. MongoDB is written in C++ and features auto-sharding and is designed with scalability and high availability in mind.

MongoDB uses dynamic schemas, which allows us to not define the structure first and still being able to create records. You can change the structure of documents simply by deleting existing fields or adding new ones.

By supporting range queries and regular expression searches, MongoDB allows ad hoc queries. Documents in a collection do not need to have same type of information. MongoDB is designed around the principle that data isn't always the same.

© Springer Nature Singapore Pte Ltd. 2018
M. Singh et al. (Eds.): ICACDS 2018, CCIS 906, pp. 55–65, 2018.
https://doi.org/10.1007/978-981-13-1813-9_6

Example of insert, find and update queries in MongoDB.

```
db.users.insert({
  user_id: 'bcd001',
  weight: 85,
  status: 'C'
})
db.users.find()
db.users.update(
  { weight: { $gt: 65 } },
  { $set: { status: 'A' } },
  { multi: false }
)
```

2 Literature Survey

This section of the paper surveys many research papers that what are the problems that MongoDB faces and how resilient data structure can help for it.

[1] proposes the resilient SDN (Software Defined Network) based architecture for Industrial Control Networks and it portrays several SDN based fast technologies. In [2] a design theory is developed for resilient software intensive-systems because of which different communities having different technologies can share a common frame of reference and knowledge (Fig. 1).

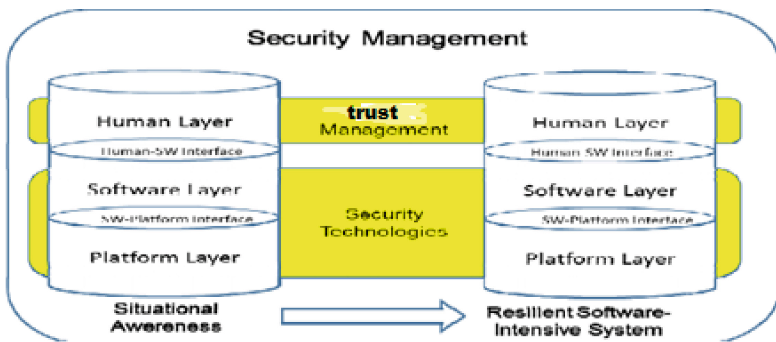

Fig. 1. Resilient software intensive-systems

In [3] Persistency-of-communication (PoC) is been introduced that deals with the communication failure problem of self triggered consensus network. [4] Provides jest of resilient software defined networks for technology related disasters. In [5] focus is mainly on the automatic load balancing of the MongoDB and proposes heat based load balancing at a very low cost (Fig. 2).

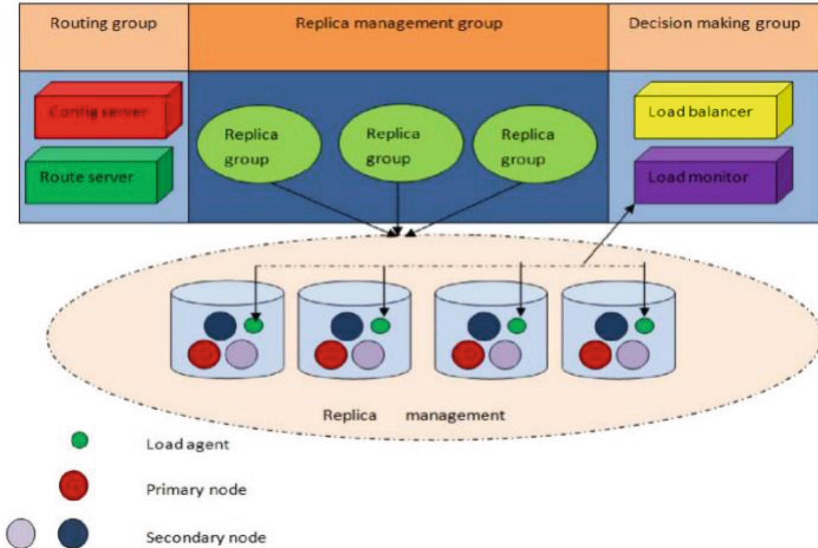

Fig. 2. Load balancing in MongoDB

In [6] modeling style of MongoDB or NoSql is discussed that if normalization is extended as it has less space for document or collection, then how it reduces the query execution time without using join operations. [7] Provides the modeling of the MongoDB with the relational algebra and also analyses the feasibility of moving from SQL database to NoSql. It does provide the optimization of the MongoDB.

[8] Proposes a method which can achieve robustness to transmission errors of Dirac which is an open source video codec (Fig. 3).

[9] Proposes an algorithm that detects fault tolerant termination based of Dijkstra that is previous fault sensitive scheme. [10] Proposes a monitor that can integrate any MongoDB deployment with the help of easy configuration changes results shows the it requires low overhead access. [11] Shows that a document based on NoSql (MongoDB) which uses JSON data and how it explores the merits of NoSql Databases (Fig. 4).

[12] Bhamra has presented a study on various type of distributed NoSql and has compared them based on various performance parameters like scalability, availability, consistency and partitioning tolerance named as CAP theory. Study shows various aspects to select a correct technology of NoSql for your application. The parameters are as follows (Fig. 5):

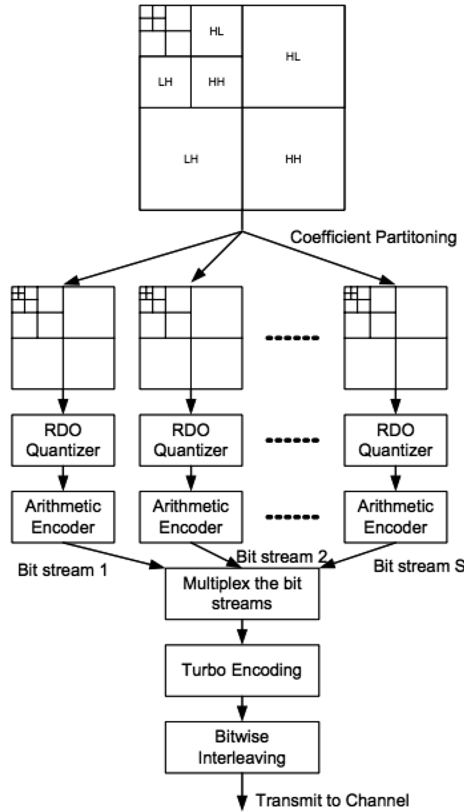

Fig. 3. Video codec method

Data model, Security, Consistency model, Ease of use, Available APIs, Platform dependency, CAP classification, Query model, Available resources such as documentation and community.

3 Proposed Work

This Paper will give you deep understanding on MongoDB concepts needed to deploy a resilient database.

We're going to do UpdateOne resiliently:

```
updateOne({'_id': '2016-06-28'},
          {'$inc': {'counter': 1}},
          upsert=True)
```

Fig. 4. MongoDB security model

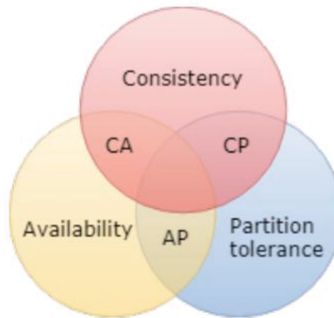

Fig. 5. CAP classification

The operation counts the number of times an event occurred, by incrementing a field named “counter” in a document whose id is today’s date. Pass “upsert = True” to create the document if this is the first event in the day.

We identify two errors that could occur while sending our message to MongoDB. These are errors bring contradiction to the data. Resilient algorithms will allow us to eliminate these errors and bring atomicity to our program.

3.1 Transient Errors

When we send his updateOne message to MongoDB, the driver may see a transient error from the network layer, such as a TCP reset or a timeout. The driver cannot tell if the server received the message or not, so we don't know whether his counter was incremented. There are other transient errors that look the same as a network blip. If the primary server goes down, the driver gets a network error the next time it tries to send it a message. This error is brief, since the replica set elects a new primary in a couple seconds. Similarly, if the primary server steps down (it's still functioning but it has resigned its primary role) it closes all connections. The next time the driver sends a message to the server it thought was primary, it gets a network error or a "not master" reply from the server.

Persistent Errors
There might also be a lasting network outage. When the driver first detects this problem it looks like a blip: the driver sends a message and can't read the response. Again, we cannot tell whether the server received the message and incremented the counter or not.

3.2 Command Errors

When the driver sends a message, MongoDB might return a specific error, saying that the command was received but it could not be executed. Perhaps the command was misformatted, the server is out of disk space, or the application isn't authorized.

Handling the Errors
The Server Discovery and Monitoring Spec requires a MongoDB driver to track the state of each server it's connected to. This data structure is called the "topology description". If there's a network error while talking to a server, the driver sets that server's type to "Unknown", then throws an exception. The topology description is:

Server 1: Unknown
Server 2: Secondary
Server 3: Secondary

In the case of a command error, what the driver thinks about the server hasn't changed: if the server was a primary or a secondary, it still is. Thus the driver does not change the state of the server in its topology description, it just throws the exception.

Code that is about resilience tries 5 times or even 10 times.

```
i = 0
while True:
    try:
        do_operation()
        break
    except network error:
        i += 1
        if i == MAX_RETRY_COUNT:
            throw
```

In the case of a network blip, we no longer risks undercounting. Now we risk overcounting, because if the server read our first updateOne message before we got a network error, then the second updateOne message increments the counter a second time.

During a persistent outage, retrying more than once wastes time. After the first network error, the driver marks the primary server “unknown”; when we retry the operation, it blocks while the driver attempts to reconnect, checking twice per second for 30 s. If all that effort within the driver code hasn’t succeeded, then trying again from his code, reentering the driver’s retry loop, is fruitless.

These are errors bring contradiction to the data. Resilient algorithms will allow us to eliminate these errors and bring atomicity to our program.

4 Results

Idempotent operations are those which have the same outcome whether you do them once or multiple times. If we make all the operations idempotent, we can safely retry them without danger of over counting or any other kind of incorrect data from sending the message twice.

MongoDB has four kinds of operations: find, insert, delete, and update.

4.1 Find

Queries are naturally idempotent.

```
try:
  doc = findOne()
except network err:
  doc = findOne()
```

4.2 Insert

This insert is idempotent. The only warning is, we have assumed we have no unique index on the collection besides the one on _id that MongoDB automatically creates.

```
doc = {_id: ObjectId(), ...}
try:
  insertOne(doc)
except network err:
    try:
        insertOne(doc)
    except DuplicateKeyError:
        pass  # first try worked
    throw
```

4.3 Delete

If we delete one document using a unique value for the key, then doing it twice is just the same as doing it once.

```
try:
  deleteOne({'key': uniqueValue})
except network err:
  deleteOne({'key': uniqueValue})
```

4.4 Update

This Update is Naturally indempotent.

```
updateOne({ '_id': '2016-06-28'}, {'$set':{'sunny': True}},
upsert=True)
```

Dealing with the original non-idempotent updateOne:

```
updateOne({ '_id': '2016-06-28'},
          {'$inc': {'counter': 1}},
          upsert=True)
```

We're going to split it into two parts. Each will be idempotent, and by transforming this into a pair of idempotent operations we'll make it safe to retry.

In Part One, we leave N alone, we just add a token to a "pending" array. We need something unique to go here; an ObjectId does nicely:

```
oid = ObjectId()
try:
  updateOne({ '_id': '2016-06-28'},
            {'$addToSet': {'pending': oid}},
            upsert=True)
except network err:
    try again, then throw
```

For Part Two, with a single message we query for the document by its _id and its pending token, delete the pending token, and increment the counter.

```
try:
   # Search for the document by _id and pending token.
   updateOne({'_id': '2016-06-28',
               'pending': oid},
              {'$pull': {'pending': oid},
               '$inc': {'counter': 1}},
              upsert=False)
except network err:
    try again, then throw
```

So we can safely retry this updateOne. Whether it's executed once or twice, the document ends up the same:

```
{
  _id: '2016-06-28',
  counter: N + 1,
  pending: [ ]
}
```

The Results obtained after resiliently coding are algorithms which do not act redundantly due to error Sect. 3.1 and maintain atomicity by not being affected by error Sect. 3.2.

5 Conclusion

The Algorithm Solution introduced in this research paper, is a modified query that provides a network error reduction alternative. We presented an algorithm solution for using Resilient Programming. It does not use repetitions, which makes it more effective over a for loop error prevention method.

This superiority increases exponentially as the lengthquri of the code increases. In this paper we took queries and converted them into the proposed Resilient queries. Manually Coding for Resilience results in a zero Energy Model, which is desirable.

References

1. Vestin, J., Kassler, A., Akerberg, J.: Resilient software defined networking for industrial control networks. In: 2015 10th International Conference on Information, Communications and Signal Processing (ICICS), Singapore, pp. 1–5 (2015)
2. Rajamäki, J., Pirinen, R.: Critical infrastructure protection: towards a design theory for resilient software-intensive systems. In: 2015 European Intelligence and Security Informatics Conference, Manchester, p. 184 (2015)
3. Senejohnny, D., Tesi, P., De Persis, C.: A jamming-resilient algorithm for self-triggered network coordination. IEEE Trans. Control Netw. Syst. **PP**(99), 1 (2016)
4. Mas Machuca, C., et al.: Technology-related disasters: a survey towards disaster-resilient software defined networks. In: 2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM), Halmstad, pp. 35–42 (2016)
5. Wang, X., Chen, H., Wang, Z.: Research on improvement of dynamic load balancing in MongoDB. In: 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing, Chengdu, pp. 124–130 (2013)
6. Kanade, A., Gopal, A., Kanade, S.: A study of normalization and embedding in MongoDB. In: 2014 IEEE International Advance Computing Conference (IACC), Gurgaon, pp. 416–421 (2014)
7. Zhao, G., Huang, W., Liang, S., Tang, Y.: Modeling MongoDB with relational model. In: 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies, Xi’an, pp. 115–121 (2013)
8. Myo, T., Fernando, W.A.C.: An error-resilient algorithm based on partitioning of the wavelet transform coefficients for a DIRAC video codec. In: Tenth International Conference on Information Visualisation (IV 2006), London, England, pp. 615–620 (2006)
9. Lai, T.-H., Wu, L.-F.: An (N-1)-resilient algorithm for distributed termination detection. IEEE Trans. Parallel Distrib. Syst. **6**(1), 63–78 (1995)
10. Colombo, P., Ferrari, E.: Enhancing MongoDB with purpose-based access control. IEEE Trans. Dependable Secur. Comput. **14**(6), 591–604 (2017)
11. Jose, B., Abraham, S.: Exploring the merits of NoSQL: a study based on MongoDB. In: 2017 International Conference on Networks and Advances in Computational Technologies (NetACT), Thiruvananthapuram, pp. 266–271 (2017)
12. Bhamra, K.: A comparative analysis of MongoDB and Cassandra. Master’s thesis, The University of Bergen (2017)