CSE 512: Distributed Database Systems

Project Report: Distributed NoSQL Database Systems Implementation (Part 5)

Group Name: Data Dominators

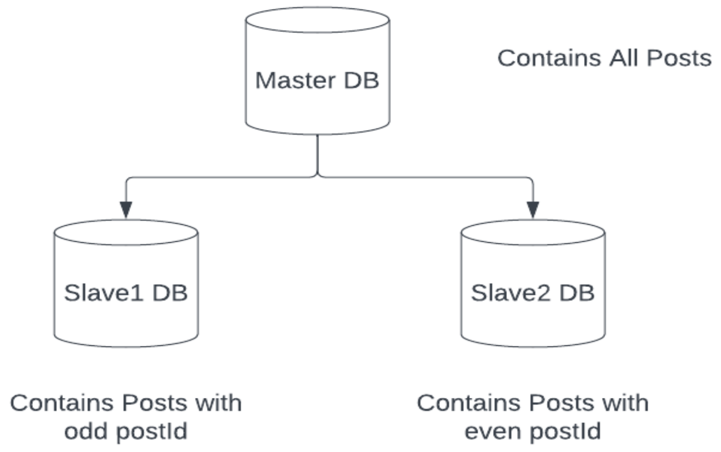
1. Introduction

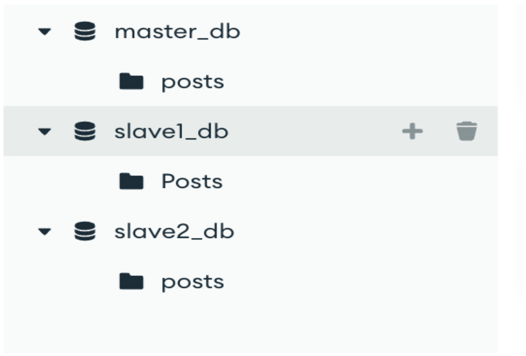
In part 5, we have implemented Distributed NoSQL database system using MongoDB. We have implemented CRUD operations, Data Distribution, Consistency, Indexing, Query Optimization, Master-Slave Replication for Posts data in a social network database.

2. Implementation

a. Master-Slave Model:

The implemented Master-Slave model consists of one master database and two slave databases. The master database, named master_db, serves as the primary data source and handles write operations. Two slave databases, slave1_db and slave2_db, act as nodes in the system and replicate data from the master database. The replication factor is set to 2, meaning each document in the master database is replicated in both slave databases.


Master DB
Contains All Posts
Slave1 DB
Slave2 DB
Contains Posts with odd postId
Contains Posts with even postId



b. **CRUD Operations**

The system supports basic CRUD (Create, Read, Update, Delete) operations for managing the **Posts** collection in each database:

i. **Create (addPost):** Inserts a new post into the master collection and replicates it to one of the slave collections based on the post's ID.

ii. **Read (getAllPosts, getAllPostsByUser, getPost):**
   - **getAllPosts**: Retrieves all posts from both slave collections and combines the results.
   - **getAllPostsByUser**: Retrieves all posts for a specific user from the master collection.
   - **getPost**: Retrieves a specific post based on its ID from the master collection.

iii. **Update (updatePost):** Updates a post in the master collection and replicates the update to the appropriate slave collection based on the post's ID.

iv. **Delete (deletePost):** Deletes a post from the master collection and replicates the deletion to the appropriate slave collection based on the post's ID.

```python
# CRUD operations
def add(collection, post):
    collection.insert_one(post)

def getAll(collection, query=None):
    return list(collection.find()) if query else list(collection.find())

def getPost(collection, postId):
    return collection.find_one({"postId": postId})

def update(collection, postId, post):
    collection.update_one({"postId": postId}, {"$set": post})

def delete(collection, postId):
    collection.delete_one({"postId": postId})

#Inserting in master and slave nodes with replication factor 2
def addPost(postId, userId, type, text, url, timestamp):
    post = Post(postId, userId, type, text, url, timestamp)
    add(master_collection, post.__dict__)
    postId = post.postId
    if(postId%2==0):
        add(slave2_collection, post.__dict__)
    else:
        add(slave1_collection, post.__dict__)

# Distributed Querying
def getAllPosts():
    list1 = getAll(slave1_collection)
    list2 = getAll(slave2_collection)
    return list1.extend(list2)

def getAllPostsByUSer(userId):
    return master_collection.find({"userId": userId})

def updatePost(postId, updatedPost):
    post = getPost(master_collection, postId)
    post["text"] = updatedPost["text"]
    post["timestamp"] = datetime.datetime.now()

    update(master_collection, postId, post)
    if(postId%2==0):
        update(slave2_collection, postId, post)
    else:
        update(slave1_collection, postId, post)

def deletePost(postId):
    delete(master_collection, postId)
    if(postId%2==0):
        delete(slave2_collection, postId)
    else:
        delete(slave1_collection, postId)
```

c. Data Distribution

Data distribution is achieved by using the addPost function to insert posts into the master collection and replicate them to the appropriate slave collection based on the post's ID. The decision to replicate to either slave1_collection or slave2_collection is determined by the post's ID. Odd post IDs are replicated to slave1_collection, while even post IDs are replicated to slave2_collection.

```
#Inserting in master and slave nodes with replication factor 2
def addPost(postId, userId, type, text, url, timestamp):
    post = Post(postId, userId, type, text, url, timestamp)
    add(master_collection, post.__dict__)
    postId = post.postId
    if(postId%2==0):
        add(slave2_collection, post.__dict__)
    else:
        add(slave1_collection, post.__dict__)
```

d. **Query Optimization**

The system incorporates query optimization through the creation of indexes on the postId field for each collection. Indexes are created on master_collection, slave1_collection, and slave2_collection to enhance the performance of queries that involve searching or sorting based on the postId field.

And we distribute the read load to the slaves so that the load is shared between nodes which improves the performance significantly.

```
# Distributed Querying
def getAllPosts():
    list1 = getAll(slave1_collection)
    list2 = getAll(slave2_collection)
    return list1.extend(list2)
```

```
# Creating indexes for query optimization and performance
master_collection.create_index([("postId", pymongo.ASCENDING)])
slave1_collection.create_index([("postId", pymongo.ASCENDING)])
slave2_collection.create_index([("postId", pymongo.ASCENDING)])
```

e. **Consistency**

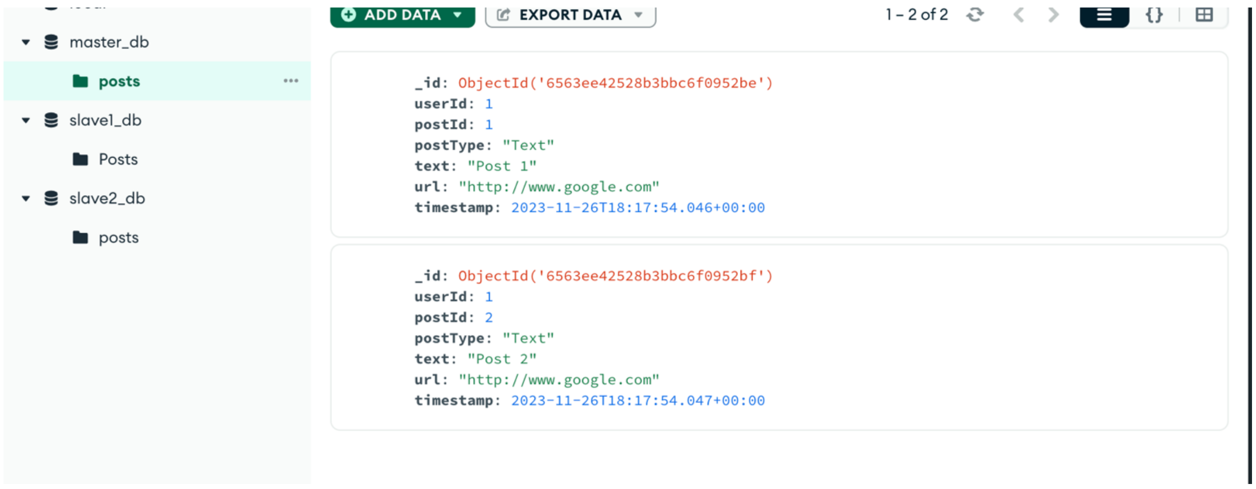
Consistency is maintained through the restore_consistency function. This function checks the existence of the master and slave databases and creates new databases if any of them is missing. The consistency is restored by copying data from the existing databases (master_db, slave1_db, slave2_db) to newly created databases (new_master_db, new_slave1_db, new_slave2_db) as needed. This ensures that each database has a consistent set of data, and the replication factor is maintained even if a database is missing. The checkReplicaSetConsistency function is used to verify the consistency of the replica set.

```
# Fault tolerant, Check if all expected databases are present in system
def checkReplicaSetConsistency(client):
    cur_all_databases = client.list_database_names()
    expected_databases = ["master_db", "slave1_db", "slave2_db"]
    print(cur_all_databases)
    return all(db in cur_all_databases for db in expected_databases)

def restore_consistency(client):
    master_db_name = "master_db"
    slave1_db_name = "slave1_db"
    slave2_db_name = "slave2_db"
    cur_all_databases = client.list_database_names()
    # Check if master_db is alive, If master_db is not alive, create a
new master_db and insert data from both slave databases
    if master_db_name not in cur_all_databases:
        print("Master")
        new_master_db = client[master_db_name]
        for document in client[slave1_db_name].Posts.find({"postId":
{"$mod": [2, 1]}}):
            new_master_db.posts.insert_one(document)
        for document in client[slave2_db_name].Posts.find({"postId":
{"$mod": [2, 0]}}):
            new_master_db.posts.insert_one(document)

        print("Restored consistency: Created new master_db")

    # Check if slave1_db is alive, If slave1_db is not alive, create a
new slave1_db and insert data with odd postId from master_db
    if slave1_db_name not in cur_all_databases:
        print("Slave 1")
        new_slave1_db = client[slave1_db_name]

        for document in client[master_db_name].Posts.find({"postId":
{"$mod": [2, 1]}}):
            new_slave1_db.posts.insert_one(document)

        print("Restored consistency: Created new slave1_db")

    # Check if slave2_db is alive, If slave2_db is not alive, create a
new slave2_db and insert data with even postId from master_db
    if slave2_db_name not in cur_all_databases:
        print("Slave 2")
        new_slave2_db = client[slave2_db_name]
        docs = client[master_db_name].Posts.find({"postId": {"$mod": [2,
0]}})
        for document in docs:
            new_slave2_db.posts.insert_one(document)

        print("Restored consistency: Created new slave2_db")
```

3. Conclusion

In summary, the implemented Master-Slave architecture leverages a master database (master_db) and two slave databases (slave1_db and slave2_db) to achieve data replication with a replication factor of 2. CRUD operations are efficiently supported, with the addPost function ensuring balanced data distribution among slave nodes based on post IDs. Query optimization is implemented through index creation on the postId field, enhancing query performance. Consistency is a central concern, addressed by the restore_consistency function, which verifies and restores a consistent state across databases. This approach ensures fault tolerance and scalability, making the system well-suited for diverse workloads and maintaining data integrity across distributed nodes.

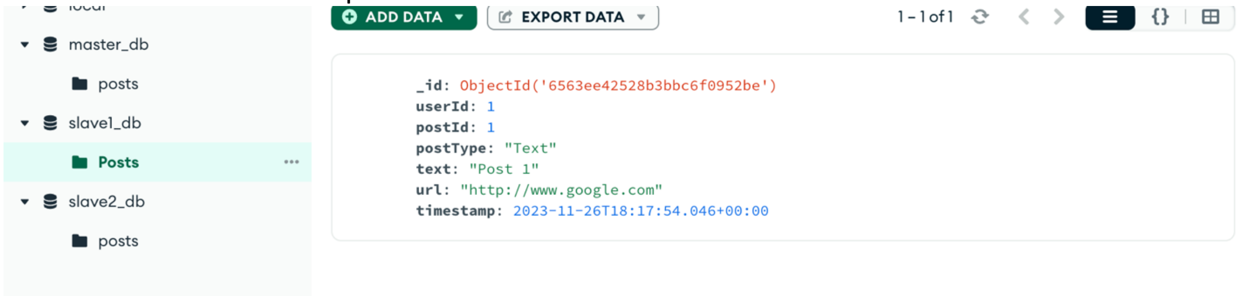
Screenshots after adding data distributed to slave databases:
Master DB:



```
_id: ObjectId('6563ee42528b3bbc6f0952be')
userId: 1
postId: 1
postType: "Text"
text: "Post 1"
url: "http://www.google.com"
timestamp: 2023-11-26T18:17:54.046+00:00
```

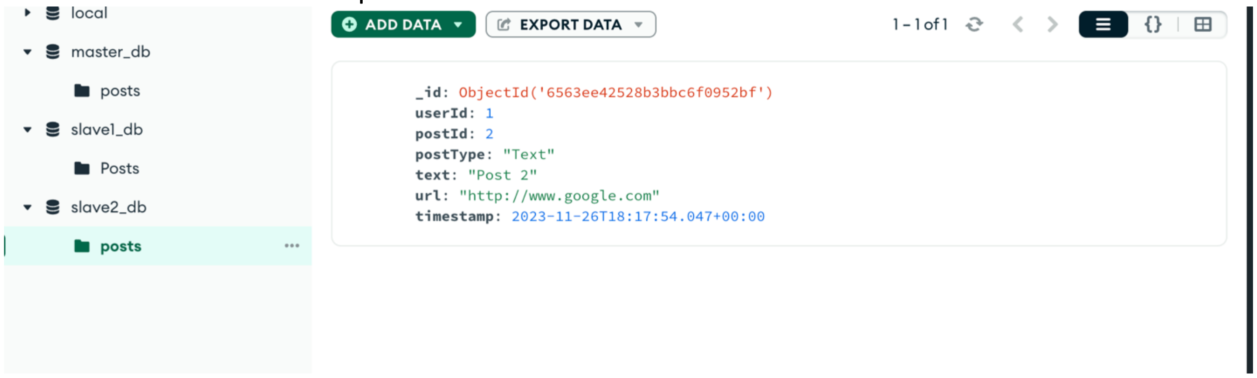
```
_id: ObjectId('6563ee42528b3bbc6f0952bf')
userId: 1
postId: 2
postType: "Text"
text: "Post 2"
url: "http://www.google.com"
timestamp: 2023-11-26T18:17:54.047+00:00
```

Slave1 DB: Posts with odd postIds



```
_id: ObjectId('6563ee42528b3bbc6f0952be')
userId: 1
postId: 1
postType: "Text"
text: "Post 1"
url: "http://www.google.com"
timestamp: 2023-11-26T18:17:54.046+00:00
```

Slave2 DB: Posts with even postIds



```
_id: ObjectId('6563ee42528b3bbc6f0952bf')
userId: 1
postId: 2
postType: "Text"
text: "Post 2"
url: "http://www.google.com"
timestamp: 2023-11-26T18:17:54.047+00:00
```

As you can see that the replication factor is 2,

- Frequency of document with postId 1(odd):
  - Master DB - 1
  - Slave1 DB – 1
- Frequency of document with postId 2(even):
  - Master DB - 1
  - Slave1 DB - 1